MASTER EN SCIENCES INFORMATIQUES

Mémo-progrès
compilateur du langage LCM-étendu

Thelen, Dominique

Award date:
1979

Awarding institution:
Universite de Namur

M E M O - P R O G E S

COMPILATEUR DU LANGAGE LCM - ETENDU

Au terme de ce travail, je tiens à remercier Monsieur A. Clarinval, promoteur de ce mémoire, pour l'aide et les encouragements qu 'il m'a prodigués au long de sa réalisation.

Je remercie Dominique qui a remarquablement mis en oeuvre ce mémoire.

Enfin, je tiens à marquer ma profonde reconnaissance à Anne-Marie pour la discrète et non moins profonde compréhension qu'elle m'a témoignée tout au long de ce travail.

PLAN

INTRODUCTION

MEMO-PROGES est une "méthode modulaire de programmation pour l'informatique de gestion". (1)

Elle distingue deux classes de modules programmés : les "fonctions intermédiaires" et les "fonctions terminales". L'interface entre deux fonctions (ensemble des données communiquées d'un module à l'autre) forme un "fichier virtuel".

Les fonctions intermédiaires, définies sur les fichiers virtuels, sont décrites au moyen du "Langage de Description de Fonctions" (LDF). Ce langage comprend :

- une clause de déclaration de fonction,
- des clauses de déclaration des structures de données dans les fichiers virtuels,
- des instructions d'accès aux fichiers virtuels (lesquelles seront traduites sous la forme d'instructions de communication entre modules),
- des primitives de structuration des traitements : titres de sections et quantificateurs de traitements,
- une fonction de copie implicite des données.

Un tel langage est incomplet. Un texte LDF doit donc contenir, notamment pour la programmation des opérations sur les données, des "séquences terminales" rédigées dans un autre langage. Ce "langage terminal" sera, au stade de l'analyse, un pseudo-code quelconque et, au stade suivant, un langage de programmation. Les séquences terminales pourront également contenir des instructions du "Langage de Contrôle Modulaire" (LCM) pour la manipulation des mécanismes de contrôle standardisés du système de programmation MEMO-PROGES.

Muni de quelques extensions, le LCM pourra également être utilisé dans la rédaction des autres modules itératifs - principalement les fonctions terminales d'accès aux fichiers réels - requis pour la réalisation d'une unité de traitement exécutable.

Les éléments du LCM doivent être considérés comme formant une extension du langage terminal, et comme appartenant à celui-ci.

Nous présentons ici la réalisation d'une première version d'un compilateur du langage LCM-étendu, produisant un texte objet en COBOL, lequel devra ensuite être compilé par le compilateur COBOL de l'installation.

Le langage COBOL retenu pour le texte objet est entièrement conforme à la norme ANS-COBOL 1974 et accepté par le compilateur COBOL du DEC System-20. Le pré-compilateur LCM est lui-même rédigé dans le même langage COBOL. Seuls, les modules terminaux d'accès aux fichiers du compilateur utilisent des éléments particuliers à la version DEC du langage COBOL.

CHAPITRE I : Le langage LCM

A. Définitions syntaxiques

B. Compatibilité syntaxique avec COBOL

C. Spécifications détaillées.

Chapitre I : Le langage "LCM"

Le langage LCM-étendu - "Langage de contrôle modulaire " (2) est ici présenté dans une syntaxe adaptée, permettant son insertion dans un texte COBOL. Le lecteur est supposé posséder les rudiments du langage COBOL.

A. Définitions syntaxiques

1. Symboles atomiques

Dans le texte d'un programme, les éléments suivants sont sensés constituer des symboles atomiques, indécomposables : les mots, les littéraux, les entiers, les signes spéciaux. Les mots, littéraux, entiers, peuvent être coupés, s'étendre sur plusieurs lignes conformément aux règles du langage COBOL.

a. Signes spéciaux

LCM utilise les signes spéciaux suivants :

- le caractère "espace".
- les caractères "=", "(", ")", " à ".
- les signes de ponctuation du langage COBOL "." , "," , ";".

Règles de ponctuation : les signes de ponctuation ";" et "," , comme en COBOL, n'ont aucune valeur sémantique. Ils sont toujours facultatifs et interchangeables. Chaque fois qu'un point apparaît dans une définition syntaxique, sa présence est obligatoire dans le texte.

b. Littéraux

Un littéral est un littéral COBOL alphanumérique, c'est-à-dire une chaîne de un à 120 caractères quelconques, comprise entre guillemets (").

c. Entiers

Un entier est un entier COBOL, c'est-à-dire une chaîne de 1 à 18 chiffres décimaux, non signés et de valeur strictement positive.

d. Mots

Un mot est un mot COBOL valide, c'est-à-dire :

- une chaîne de un à 30 caractères.
- choisis parmi les suivants : les lettres "A" à "Z"
  les chiffres "0" à "9"
  le tiret "-"
- contenant au moins une lettre en position quelconque
- dont le premier et le dernier caractère ne sont pas un tiret.

2. Classes de mots

Trois classes de mots sont définies en LCM : les mots clés, les noms de catalogue et les identificateurs.

a. Mots clés

Les mots clés sont des mots définis par la syntaxe elle-même du langage; ils permettent la reconnaissance syntaxique des phrases du langage.

b. Identificateurs

Un identificateur est un mot défini par le programmeur pour désigner une réalisation d'objet dans le programme. Un identificateur doit être un mot COBOL valide, qui peut être qualifié, indexé , ou indicé.

c. Noms de catalogue

Un nom de catalogue est un mot défini par le programmeur pour identifier une description de type d'objet enregistrée dans les catalogues du système "MEMO-PROGES".
Tout nom de catalogue doit être univoque, désigner une seule description dans l'ensemble de toutes les descriptions cataloguées dans le système.

d. Règles d'emploi

Un mot clé ne peut , dans le texte du programme, figurer qu'aux endroits où le montrent les définitions syntaxiques.

L'ensemble des identificateurs déclarés dans un programme (que leur déclaration soit faite par une clause LCM ou COBOL) et le sous-ensemble des noms de catalogue référencés doivent être disjoints.
L'innobservance de cette règle pourrait rendre impossible la programmation de références univoques.

3. Constructions mixtes

La définition syntaxique du LCM fait appel à trois concepts : séquence, pseudoséquence et référence, qui sont des constructions contenant à la fois des éléments du langage LCM et des éléments du langage terminal COBOL. Ces constructions ne sont que partiellément analysées par le précompilateur LCM.

a. Séquences

Une séquence est une suite d'instructions, pouvant contenir des étiquettes de paragraphes COBOL.
Les traitements programmes à l'intérieur d'une séquence sont exécutés dans l'ordre où ils sont rédigés. La succession chronologique des séquences est réglée par les mécanismes du contrôle dynamique de la méthode MEMO-PROGES. Il existe différents types de séquences :

- séquences terminales :
  = séquences pouvant contenir : - toute procédure COBOL
    - toute instruction et tout **test** LCM.
- séquences terminales à restriction :
  = séquences ne pouvant contenir aucune instruction de communication entre modules (instructions à OPEN, à ICALL, à CLOSE)

Règle : Une séquence ne peut contenir aucune instruction de branchement (GO ou PERFORM) sur une procédure située en dehors d'elle.
Exception : Dans toute séquence, une instruction PERFORM peut citer une procédure située dans la section à SUBPARTS.
(L'observance de cette règle n'est pas vérifiée par le pré-compilateur LCM, puisque celui-ci n'analyse pas

l'instruction PERFORM du langage COBOL. Elle n'est pas non plus vérifiée par le compilateur COBOL, celui-ci ne connaissant pas le concept de séquence).

b. Pseudo-séquences

Nous appelons pseudo-séquence une partie de programme possédant la syntaxe d'une séquence terminale, mais non pas sa sémantique : l'exécution d'une pseudo-séquence est commandée par les mécanismes propres au langage COBOL plutôt que par les mécanismes de contrôle du système "MEMO-PROGES".

c. Références

Une référence est une référence COBOL valide à une donnée. Il s'agit d'un identificateur simple, qualifié, indicé , ou indexé.
(Le précompilateur LCM n'analysant pas la DATA DIVISION du programme COBOL ne peut vérifier la validité des références. Cette vérification est différée jusqu'à la compilation COBOL).

4. Types de phrases

Les plus petites constructions significatives dans le LCM sont appelées "phrases". On distingue différents types de phrases : les déclarations, titres, instructions, tests et noms de variables.

a. Déclarations

Les déclarations LCM permettent de copier dans le texte du programme la description d'une structure de données enregistrée dans les catalogues du système MEMO-PROGES. Le système de catalogues n'étant pas actuellement réalisé, la version du LCM présentée ici ne comporte pas ces déclarations.

b. Titres

Un certain nombre de titres sont prévus dans le texte d'un programme COBOL/LCM. Ils définissent la structure du programme en le découpant en segments et permettent la gestion implicite de la dynamique correspondant à cette structure.

Les règles syntaxiques des titres COBOL sont applicables :

- le premier mot doit commencer en zone A de la ligne COBOL; les autres mots doivent figurer en zone B de la ligne; un titre doit se terminer par le séparateur ".".
- un titre ne peut s'étendre sur plusieurs lignes.
- une ligne comportant un titre ne peut contenir aucun autre texte.

c. Instructions

Une instruction est une phrase prescrivant une opération. Une instruction doit figurer en PROCEDURE DIVISION à tout endroit de la zone B de la ligne. Toute phrase COBOL dans la procédure division peut contenir à la fois des instructions (propositions) COBOL et des instructions LCM.

d. Tests

En COBOL, une conditions est une combinaison de tests introduite par une des conjonctions IF, UNTIL, WHEN . COBOL fournit différents types de tests.
Le LCM propose un nouveau type de tests portant sur la valeur de certains indicateurs de contrôle du système MEMO-PROGES.
Une instruction COBOL introduite par IF ou UNTIL ne peut entre autre tests contenir des tests LCM.

e. Noms de variables

Certaines variables, accessibles aux instructions (particulièrement les instructions COBOL) sont implicitement déclarées dans tout programme COBOL-LCM. Elles sont définies par des mots clés du langage. (Ces variables sont assimilables aux registres spéciaux COBOL).

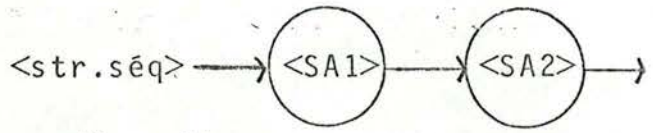
5. Structures dans les phrases LCM

La définition des phrases du LCM fait appel à quatre types de structures : séquentielle, alternative, facultative et itérative. Les structures permettent de définir des relations "précédent-suivant"pour les symboles atomiques d'une phrase.

a. Structure séquentielle

La structure séquentielle définit la relation "un précédent - un suivant" pour deux symboles atomiques.

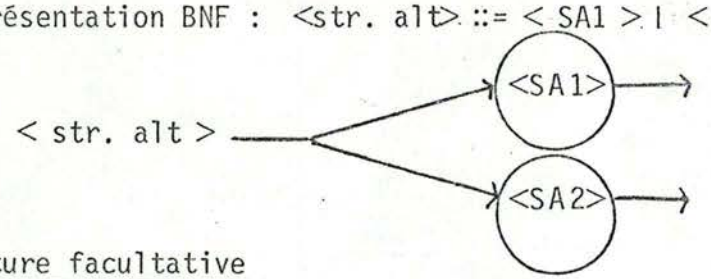
- Représentation BNF : < Str.Séq > ::= < SA1 > < SA2 >
- Représentation en "graphe syntaxique" :



b. Structure alternative

La structure alternative définit la relation "un précédent - un parmi plusieurs suivants possibles".

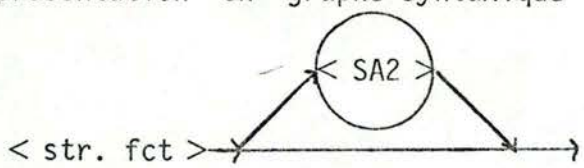
- Représentation BNF : <str. alt> ::= < SA1 > | < SA2 >



c. Structure facultative

La structure facultative définit la relation "un précédent - un suivant facultatif".

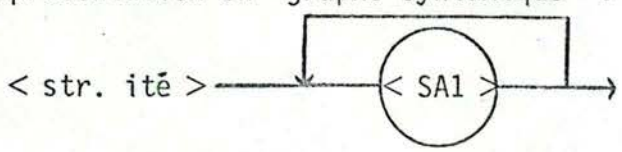
- Représentation BNF : < str. fct > ::= < SA1> [< SA2>]
- Représentation en "graphe syntaxique" :



d. Structure itérative

La structure itérative définit la relation "un précédent, un ou plusieurs suivants du même type".

- Représentation BNF : < str. ité > ::= < SA1 > [< SA2 >]
- Représentation en "graphe syntaxique" :



e. Constructions de phrases LCM

Pour chaque phrase du langage, nous définissons une syntaxe (cfr. annexe). Cette syntaxe peut être formalisée à l'aide de la représentation BNF ou de la représentation en graphe

syntaxique . Nous préfèrerons cette dernière, celle-ci étant plus proche du mécanisme d'analyse syntaxique.

A chaque phase nous associons un graphe syntaxique, dans lequel les arcs représentent les quatre types de relation précités.
Les sommets peuvent être de trois types :

1. Pour tout graphe, il existe un et un seul sommet "entrée" (sommet sans précédent). Ce sommet définit le premier symbole atomique de la phrase.

2. Il existe au moins un sommet "sortie" (sommet sans suivant). Ce sommet ne définit aucun symbole atomique : il s'agit d'un sommet "fictif" n'ayant d'autre signification que d'indiquer que la phrase est terminée.

3. Il peut exister des sommets intermédiaires (ayant un précédent et un suivant). Parmi ceux-ci, nous distingons :
   a. les sommets définissant un symbole atomique de la phrase.
   b. les sommets "de règle" . Ceux-ci définissent un sous-graphe syntaxique (cf. symbole métalinguistique dans la représentation BNF). Ceux-ci possèdent la même structure que les graphes syntaxiques : ils ne sont qu'une "abréviation" de la syntaxe.

B. Compatibilité syntaxique avec COBOL

1. Adaptation syntaxique du LCM

a. Dans un texte du programme COBOL/LCM, il est nécessaire de distinguer les phrases appartenant au LCM de celles qui appartiennent au langage COBOL. En effet, ces dernières ne sont en principe pas examinées par le précompilateur LCM. A cette fin, tout premier mot d'une phrase LCM est préfixé par le caractère "à" (qui n'est pas un caractère autorisé en COBOL).

b. Une phrase LCM, peut à l'instar des phrases COBOL, s'étendre sur plusieurs lignes.

c. Des lignes de commentaire ("-", ou "/" en colonne 7) peuvent figurer à tout endroit du texte.

d. Les signes de ponctuation ".", ";" et "," doivent être utilisés conformément aux règles du langage COBOL : un signe de ponctuation doit être suivi d'au moins un espace ou être le dernier caractère de la ligne; il peut être précédé d'un ou plusieurs espaces.

2. Règles spéciales des séquences terminales en COBOL

a. Aucun mot clé du LCM n'est un mot réservé pour le langage COBOL (seuls les mots préfixés par "à" sont réservés, mais ce ne sont pas des mots COBOL valides); en revanche, les mots réservés de COBOL peuvent être utilisés dans une phrase LCM. Les mots créés par le précompilateur dans le texte objet commencent tous par une lettre suivie d'un tiret. Le programmeur évitera d'employer lui-même de tels mots.

b. Afin de ne pas perturber le contrôle dynamique implicite des programmes, le programmeur ne peut, en guise de noms de procédures, définir que des noms de paragraphes; il ne peut définir aucun nom de section (exception faite pour les sections déclaratives - sections USE).

c. Le texte d'un programme LDF étant structuré en blocs, le programmeur pourrait éprouver des difficultés à localiser dans ces blocs certaines procédures qu' il souhaite exécuter par PERFORM. Il pourra placer ces procédures à l'intérieur de la section à SUBPARTS spécialement définie à cette fin.

C. Spécifications détaillées

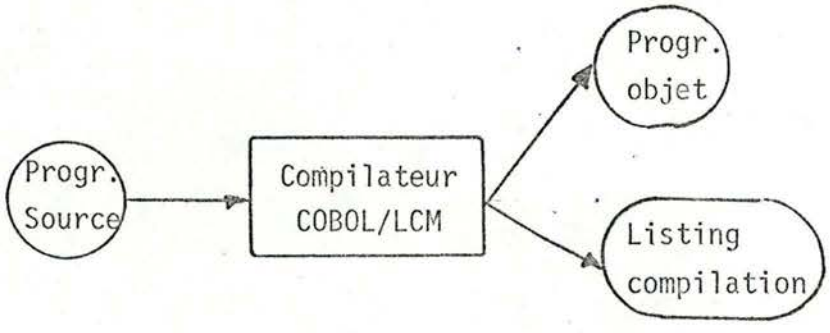
Des spécifications détaillées définissant les catégories syntaxiques, la syntaxe et l'utilisation du langage LCM sont reproduites en annexe dans un document extrait de "MEMO-PROGES. Définition des langages."

Chapitre II : Principes et méthodes de compilation.

A. Analyse lexicale
B. Analyse syntaxique
C. Analyse sémantique
D. Production de messages
E. Etablissement de la structure du programme
F. Génération du texte-objet.

Chapitre II : Principes et méthodes de compilation.

La compilation d'un programme COBOL/LCM consiste, à partir de celui-ci, à produire un programme-objet COBOL et un "listing-compilation".



Dans ce chapitre, nous présenterons les cinq pôles de réflexion qui furent les nôtres lors de l'élaboration du compilateur, à savoir : l'analyse lexicale, l'analyse syntaxique, l'analyse sémantique, la production de messages, l'établissement de la structure du programme, la génération du texte-objet. (6)(11)

A. Analyse lexicale (6) (8)

1. Buts de l'analyse lexicale

L'analyse lexicale consiste à repérer dans le texte du programme source les symboles atomiques.A chaque symbole est associé un préfixe décrivant sa structure syntaxique; les informations contenues dans ce préfixe définissent la catégorie syntaxique du symbole (mot-clé, identificateur, littéral, entier, signe spécial, symbole invalide) et sa position dans le texte source.
Ne sont pas considérés comme symboles atomiques les séparateurs "espace", ",", "et", qui n'ont pas de signification dans un programme COBOL.

2. Principes d'analyse lexicale

a. Nous avons dissocié l'analyse lexicale de l'analyse syntaxique pour plusieurs raisons :
   - les structures lexicographique et syntaxique du langage ont des "stabilités" différentes : la structure lexicographique est définie pour une installation donnée (un compilateur COBOL), la structure syntaxique est définie pour une "version" donnée du langage (celui-ci pouvant être sujet à des modifications et des extensions).
   - la syntaxe du LCM est définie à partir de symboles atomiques
   - la dissociation de ces deux problèmes permet de mieux les aborder.

b. L'analyseur lexical est considéré comme une "sous-routine" de l'analyseur syntaxique. L'analyseur syntaxique appelle l'analyseur lexical en lui demandant le symbole suivant dans le programme-source [GET-TOKEN] ; l'analyseur lexical lui "renvoie" un "TOKEN" [TOKEN] (symbole et préfixe descriptif).

Un symbole atomique pouvant être coupé et s'étendre sur plusieurs lignes, le module de lecture d'une ligne de tête est une sous routine appelée par l'analyseur lexical.

B. Analyse syntaxique (6)(9)

1. Buts de l'analyse syntaxique

L'analyse syntaxique ne porte pas sur le programme complet, mais séparément sur chaque phrase LCM rencontrée. Pour toute phrase, l'analyse comporte deux parties : la vérification de l'organisation des symboles atomiques et du "contexte syntaxique" de la phrase, c'est-à-dire à la fois sa position dans la ligne COBOL et dans les différentes parties ou segments du programme (divisions et sections).

On notera les particularités suivantes :
Les phrases COBOL ne sont pas analysées; elles sont simplement recopiées dans le programme-objet. Toutefois, les titres de divisions et de sections définis par la norme COBOL sont analysés au même titre que les titres de sections LCM; comme eux, en effet, ils définissent dans le programme des segments dont la reconnaissance est nécessaire pour l'analyse de contexte évoquée ci-dessus.
Le langage définit des relations obligatoires,possibles et interdites entre les différentes instructions citant un module (instructions à ICALL, à OPEN, à CLOSE, à PREPARE). Les relations ne peuvent être vérifiées qu'une fois la lecture du programme-source complet achevée.

2. Principes d'analyse syntaxique (10)

L'analyseur syntaxique est une sous-routine appelée par le module principal (LCMCPL) du compilateur, chaque fois qu'il détecte le début d'une phrase à analyser, c'est-à-dire soit le caractère spécial " à " précédant une phrase LCM, soit le premier mot d'un titre COBOL. Le retour au module appelant (LCMCPL) s'effectue lorsque l'analyseur a détecté la fin de la phrase analysée et après qu'il ait provoqué l'exécution de tous les traitements associés à cette phrase.

a. Descripteurs d'analyse syntaxique

La structure syntaxique du langage est mémorisée dans un fichier contenant un ensemble de descripteurs d'analyse. Ces descripteurs contiennent une représentation des algorithmes d'analyse et de traitement des différentes phrases du langage.
Nous avons préféré mémoriser les algorithmes d'analyse et de traitement sous forme de fichier, plutôt que sous forme d'algorithmes proprement dits : ce type de démarche nous donne, en effet, une relative indépendance vis-à-vis des modifications et extensions du langage.
Pour chaque phrase du langage, nous définissons un ensemble de descripteurs auxquels nous accédons au moyen du premier symbole de la phrase.
Nous distingons deux types de descripteurs : les descripteurs "ØØ" et les descripteurs "nn".

Descripteurs "ØØ"

Le descripteur "ØØ" définit le contexte autorisé pour la phrase (ensemble de segments pouvant la contenir) et les éventuels traitements particuliers à effectuer après l'analyse de celle-ci (pourvu que l'analyse ait reconnu la validité de la phrase). Ces traitements particuliers sont les suivants :

- Pour le titre à IENTRY déclarant le programme : les paramètres figurant dans ce titre doivent être mémorisés puisque la traduction de certaines phrases s'y réfère ((le titre à IENTRY doit être la première phrase LCM rencontrée dans le programme).
- Pour les titres LCM et COBOL , on doit vérifier qu'ils se rencontrent au plus une seule fois, et pour ce qui concerne les titres COBOL, dans l'ordre prévu par la norme du langage.
- Pour les titres de sections LCM, il y a lieu de clôturer dans le programme-objet la section précédente et d'ouvrir la nouvelle.
- Pour les instructions de communication entre modules, il y a lieu de créer une table des instructions rencontrées,

en vue de permettre l'analyse ultérieure des relations entre elles.

Descripteurs "nn"

Les descripteurs "nn" représentent les cheminements possibles dans le graphe syntaxique associé à chaque phrase du LCM.
A chaque sommet du graphe syntaxique, nous associons un descripteur.
Nous pouvons distinguer différents types de descripteurs :

- les descripteurs associés à l' "entrée" dans un graphe ou un sous-graphe
- les descripteurs associés à la "sortie" d'un graphe ou d'un sous-graphe
- des descripteurs associés à la description d'un élément syntaxique
- les descripteurs associés au branchement vers un sous-graphe.

I. Descripteurs associés à l'entrée dans un graphe ou dans un sous-graphe.

Un descripteur associé à l'entrée dans un graphe ou dans un sous-graphe est un descripteur du premier élément syntaxique de la phrase ou de la "règle" et est numéroté "Ø1". Il a la même structure qu'un descripteur associé à la description d'un élément syntaxique.

II. Descripteurs associés à la sortie d'un graphe ou d'un sous-graphe.

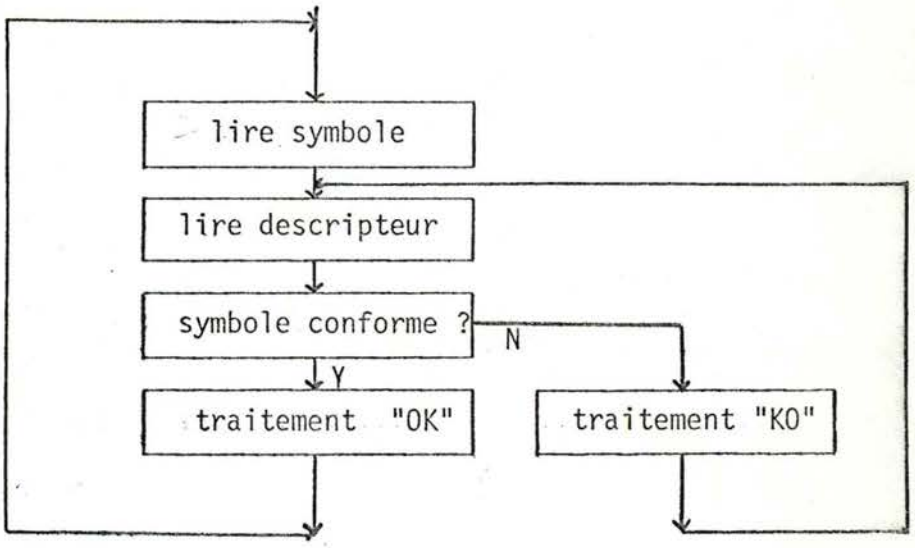
Les descripteurs associés à la sortie d'un graphe ou d'un sous-graphe sont purement "fictifs". Le dernier sommet ("fictif") du graphe est seulement référencé par les autres descripteurs.

Ce descripteur porte le numéro "99" s'il s'agit de la sortie normale du graphe (sortie sans erreur), le numéro "98" s'il s'agit de la sortie normale d'un sous-graphe. La sortie d'un graphe ou d'un sous-graphe sera numérotée "ØØ" s'il s'agit d'une sortie "erreur".

III. Descripteurs associés à la description d'un élément syntaxique.

Les descripteurs associés à la description d'un élément syntaxique contiennent les informations suivantes :

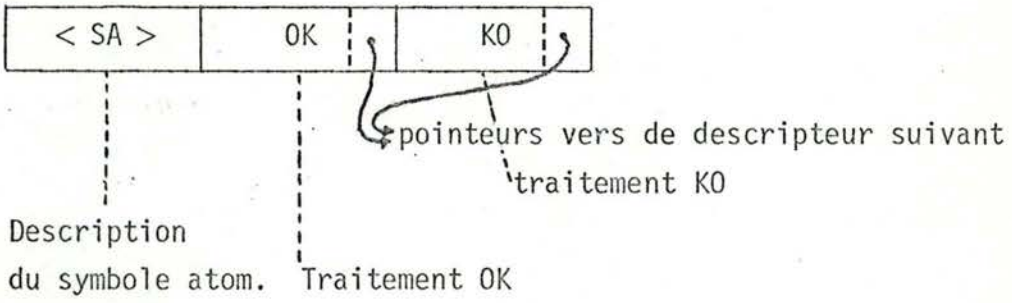
1. La description du symbole atomique attendu : sa catégorie syntaxique, sa valeur s'il s'agit d'un mot clé, le séparateur, le suivant (";" ou ",")
2. Les traitements "OK" et "KO"
   Les traitements "OK" et "KO" sont effectués selon que le symbole analysé répond ou non à la description du symbole atomique.
   Le traitement "OK" s'accompagne de la lecture du symbole suivant.



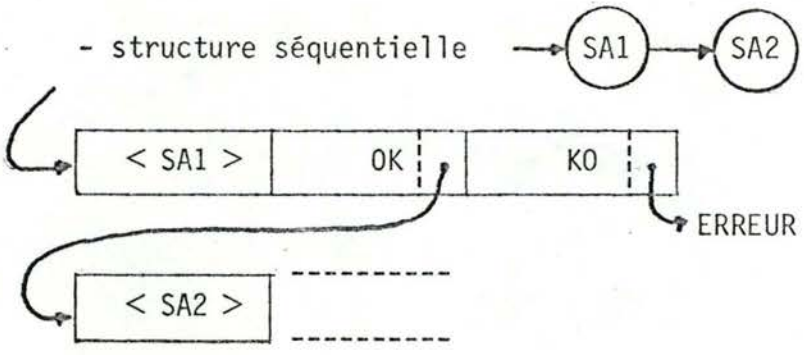
Les traitements "OK" et "KO" permettent

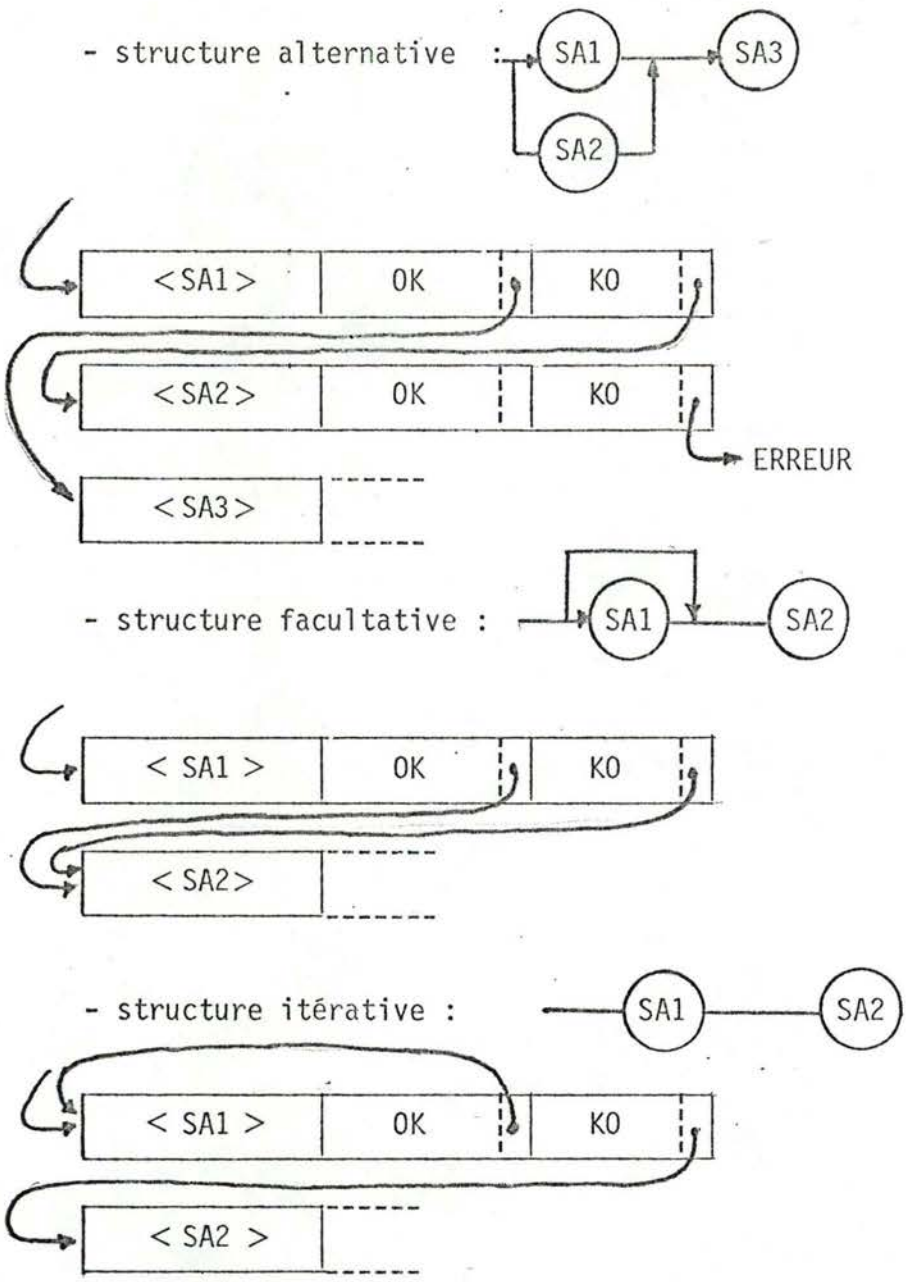
- d'envoyer un message d'erreur ou d'avertissement
- de stoker dans une table - la table des formes internes - le symbole analysé ou un texte fourni par le descripteur lui-même.
- d'identifier le descripteur suivant (pointeur) : descripteur contenant la description de l'élément syntaxique suivant, descripteur de "sortie" ou descripteur de branchement vers un sous-graphe.

Représentation d'un descripteur :



Représentation des structures syntaxiques par des descripteurs



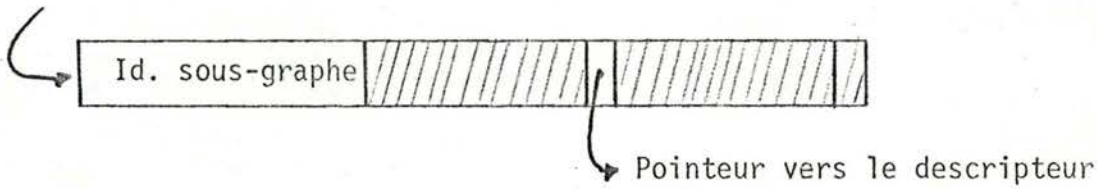


IV. Descripteurs associés au branchement vers un sous-graphe

Les sous-graphes peuvent contenir les mêmes descripteurs que les graphes syntaxiques, à l'exception des descripteurs ØØ.
Les descripteurs associés au branchement vers un sous-graphe comprennent les informations suivantes :

- l'identification du sous-graphe dans lequel se poursuivra le cheminement
- le pointeur vers le descripteur à considérer après le cheminement dans le sous-graphe.

Représentation d'un descripteur associé au branchement vers un sous-graphe.



Il convient de remarquer que le sous-graphe est une routine fermée retournant toujours au graphe appelant.

b. Production de formes internes

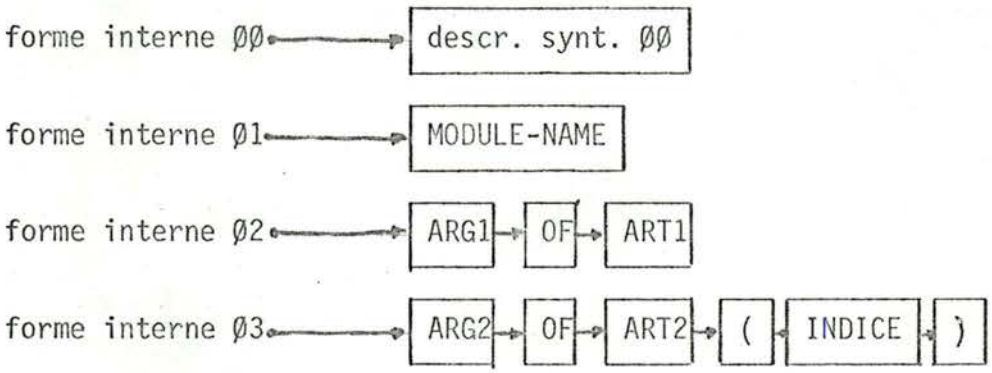
A chaque phrase LCM, nous associons une table des formes internes. Cette table constitue l'interface entre l'analyseur syntaxique (qui la crée) et le générateur de texte objet (qui la consulte). Elle contient

- soit dans une entête les informations nécessaires au traitement de la phrase (contenues dans le descripteur ØØ)
- soit la mémorisation de certains symboles ou groupes de symboles rencontrés dans la phrase analysée, de certains textes définis dans les descripteurs.

La table des formes internes est construite comme un ensemble de listes chaînées : A chaque liste correspond une forme interne.
La liste "ØØ" contient le descripteur syntaxique "ØØ".
Les listes "nn" contiennent un ou plusieurs symboles formant un tout cohérent pour le générateur de texte-objet.

Exemple : soit l'instruction : à ICALL MODULE-NAME (ARG1 OF ART1, ARG2 OF ART2 (INDICE))

Les formes internes correspondantes seront



Ces listes sont initialement vides (c'est-à-dire au début de l'analyse de la phrase).

La primitive d'insertion du producteur de formes internes est

INSERT-NEXT (numéro de forme interne)

Les primitives d'obtention du générateur de texte objet sont

OBTAIN-FIRST (numéro de forme interne)

OBTAIN-NEXT (numéro de forme interne)

C. Analyse sémantique

La version actuelle du compilateur ne comporte aucune vérification sémantique dans le programme-source.

Celle-ci consisterait à vérifier la cohérence entre données déclarées en DATA DIVISION et leur emploi en PROCEDURE DIVISION. Cette vérification demanderait l'analyse de la DATA DIVISION et la manipulation d'une table des symboles.

D'autre part, cette vérification est effectuée par le compilateur COBOL car le programme-objet conserve les références présentes dans le programme-source.

Le compilateur devrait aussi vérifier si la valeur de nombres entiers est comprise dans un ensemble de valeurs permises (instruction à INDIC (< subscript >) : où la valeur de <subscript> doit être comprise entre 1 et 120. Cette vérification n'est pas effectuée : elle demanderait la mise en place d'algorithmes spéciaux. Pour ces cas, le compilateur se contente de produire un message d'avertissement demandant au programmeur d'effectuer la vérification lui-même.

D. Production de messages

Un message à destination du programmeur peut être produit lors de l'analyse syntaxique ou lexicale.

Nous distingons deux types de messages :

- les messages d'erreurs fatales: lorsqu'une erreur est détectée, le compilateur cesse toute génération du programme-objet. Il continue néanmoins l'analyse syntaxique des phrases, mais n'effectue plus entièrement leur analyse de contexte.
- les messages d'avertissement : ils sont produits uniquement à titre d'avertissement, et n'affectent en rien la génération du programme-objet.

Mécanisme de production des messages

Le fait d'associer un message à un événement peut être mémorisé de deux façons différentes :

- soit dans le programme du compilateur
- soit dans un descripteur d'analyse syntaxique.

Lorsqu'une erreur est détectée, il s'agit de faciliter au maximum la détection et la compréhension de celle-ci par le programmeur.
Lors de la détection d'une erreur, nous créons une paire de données (numéro de message d'erreur, symbole du programme-source erroné).
Le numéro de message d'erreur permet de distinguer les erreurs fatales des avertissements et sert de clé d'accès dans un fichier contenant une explication cursive de l'erreur.
De plus, la structure de l'édition de ces messages est telle que dans le listing de contrôle de la compilation, le message d'erreur suit directement la ligne source erronée.

Exemple : à RESSET STATUS

x FATAL x MOTCLE LCM INCONNU : RESSET

type explication cursive symbole erroné

E. Etablissement de la structure du programme

1. Structure des programmes source et objet

Un programme source COBOL/LCM possède une structure dont dérive directement la structure du programme-objet. En effet, programme-source et programme-objet sont un ensemble de "segments". 'Un "segment" est défini par la présence d'un titre dans le texte :

- les titres de sections et de divisions COBOL dans l'ordre
- les titres de sections LCM (dans la PROCEDURE DIVISION) dans un ordre quelconque.

De plus, un programme COBOL/LCM, comme tout programme COBOL ne peut contenir qu'un seul segment de chaque type.

Sur cette structure, dans le programme-objet, se greffent les éléments du contrôle dynamique des programmes du système MEMO-PROGES :

- l'algorithme de contrôle de l'itération dans le programme (U-LCM)
- l'initialisation et clôture implicite des modules appelés. (U-ICALL SECTION).

PROGRAMME-SOURCE		PROGRAMME-OBJET
IDENTIFICATION DIVISION	(b)	IDENTIFICATION DIVISION (c)
àIENTRY	(c)	PROGRAMME-ID... (c)
ENVIRONMENT DIVISION	(b)	ENVIRONMENT-DIVISION (c)
DATA DIVISION	(b)	DATA DIVISION (c)
FILE SECTION	(a)	FILE SECTION (a)
WORKING-STORAGE SECTION	(a)	WORKING-STORAGE SECTION (c)
		COPY U-INT. (1)
LINKAGE SECTION	(c)	LINKAGE SECTION (c)
		COPY U-EXT. (2)
PROCEDURE DIVISION	(b)	PROCEDURE DIVISION USING U-EXT...(c)(3)
DECLARATIVES	(a)	DECLARATIVES (a)
END DECLARATIVES	(a)	END DECLARATIVES (a)
		COPY U-LCM (4)(c)
àCORPUS SECTION	(c)	U-CORPUS SECTION (c)
àCONSTANT SECTION	(a)	U-CONSTANT SECTION (c)
àFIRST SECTION	(a)	U-FIRST SECTION (c)
àLAST SECTION	(a)	U-LAST SECTION (c)
àINIT SECTION	(a)	U-INIT SECTION (c)
àTERM SECTION	(a)	U-TERM SECTION (c)
àPARAM SECTION	(a)	U-ICALL SECTION (5)

(a) facultatif
(b) facultatif mais message d'avertissement
(c) obligatoire
(1) indicateurs locaux du contrôle de l'itération
(2) indicateurs globaux du contrôle de l'itération
(3) définition de l'interface
(4) Algorithme du contrôle de l'itération
(5) initialisation et clôture implicite des modules appelés.

2. Elaboration d'une structure à partir d'un programme COBOL-LCM

La structure d'un programme-objet est élaborée sur base
- des titres COBOL et LCM
- des instructions "OPEN", "ICALL", "CLOSE", "PREPARE"

rencontrées dans le programme-source.

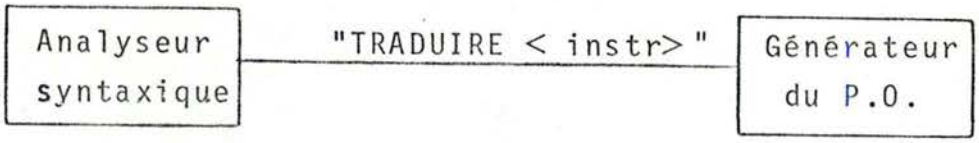
- Ceux-ci sont détectés et mémorisés lors de l'analyse syntaxique.
  - la présence d'un titre est mémorisée dans une table des segments.
  - pour les instructions "OPEN", "ICALL", "CLOSE", "PREPARE", nous mémorisons le type d'instruction et le nom du module concerné.

- L'analyse d'un titre COBOL ou LCM provoque la traduction dans un segment donné :
  nous considérons le programme-objet comme un ensemble de segments initialement vides. Le mécanisme de génération du texte-objet se chargera de remplir de manière adéquate ces différents segments.

- Après avoir terminé l'analyse du texte source nous effectuons les vérifications nécessaires dans ces deux tables et générons
  - les titres obligatoires dans le programme-objet non rencontrés dans le programme-source.
  - les initialisations et clôture implicites des modules appelés.

F. Génération du texte-objet

Le langage LCM définit pour chaque phrase sa traduction en COBOL. La traduction de la phrase peut se faire "sur place" : suite à l'analyse de celle-ci et de celles qui la précèdent. Nous devons être à même de générer sa traduction. Le générateur peut être considéré comme une "sous-routine" de l'analyseur syntaxique.



Ainsi chaque phrase est l'origine d'une ou plusieurs lignes COBOL dans le programme-objet.
La production d'une ligne COBOL nécessite certaines informations :

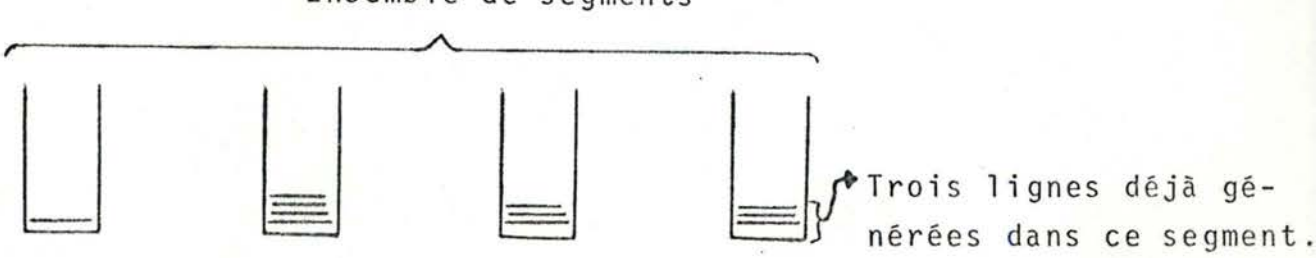
- un prototype de traduction de la ligne à générer
- la localisation de la ligne dans le programme objet.

1. Prototype de traduction

Le prototype de traduction contient le texte qui sera la traduction de la phrase LCM et des références à des symboles rencontrés dans la phrase à traduire ou dans une phrase précédemment analysée. La génération d'une ligne COBOL consistera à recopier le texte en remplaçant les références à des symboles rencontrés par les symboles eux-mêmes.
Les symboles rencontrés dans la phrase proviendront de la table des formes internes. Les symboles rencontrés dans une phrase déjà analysée proviendront de la table des "paramètres actuels".
La production d'une ligne COBOL peut être conditionnelle à l'absence ou à la présence de certaines références.

2. Localisation de la ligne générée

Chaque ligne générée doit trouver sa place dans le programme objet. Les mécanismes de génération permettent de générer une ligne dans
- le segment courant
- le segment précédent
- le segment donné.



A chaque segment, nous pouvons faire correspondre une pile : la primitive d'insertion d'une ligne objet est du type "INSERT LAST (n° pile)" (Insertion dans la ligne après celles qui se trouvent déjà dans ce segment).
Le mécanisme d'insertion dans ces différentes piles est le suivant :
Nous produisons séquentiellement toutes les lignes dans un fichier unique et à chaque ligne nous associons un préfixe indiquant
- la pile à laquelle elle appartient
- son numéro d'ordre dans la pile.

Ensuite, pour obtenir le programme-objet définitif, nous tirons ce fichier appelé programme-objet provisoire avec en guise de clé de tri le préfixe associé à chaque ligne.

CHAPITRE III : Implémentation du compilateur

A. Options générales sur la programmation
B. Structure générale du compilateur
C. Analyseur lexical
D. Analyseur syntaxique
E. Producteur du listing-compilation
F. Générateur du programme-objet

Chapitre III : Implémentation du compilateur

A. Options générales sur la programmation

1. Utilisation de la méthode MEMO-PROGES

Dans la mesure du possible, nous avons employé la méthode MEMO-PROGES lors de l'élaboration du compilateur.
Nous n'employons pas la méthode de décomposition, MEMO-PROGES n'étant pas conçu pour l'analyse d'un compilateur : en effet, celle-ci est développée pour des applications de type "gestion". En effet, pour certaines fonctions, il leur était possible de répondre à la définition proposée par la méthode, à savoir :

- posséder un et un seul fichier "maître"
- transformer un des attributs définissant l'état d'un seul de ses fichiers d'entrée.

Au niveau de la programmation des fonctions, nous avons employé la méthode MEMO-PROGES, à savoir :

- l'utilisation d'interfaces banalisés : A chaque appel de module, nous transmettons le paramètre global U-EXT contenant les indicateurs banalisés du contrôle dynamique des programmes. Lors de la programmation , nous avons remarqué qu'un certain nombre d'informations devaient être consultées ou modifiées par un grand nombre de modules : un "COMMON-BLOC" contient ses informations et est ajouté à la liste des arguments de l'instruction à ICALL de tous les modules exceptés les modules terminaux (d'accès aux fichiers).

- la gestion dynamique et des erreurs : la gestion dynamique comprend les déclarations de modules itératifs (àIENTRY), la décomposition en sections et l'instruction d'appel itératif (àICALL). La gestion des erreurs repose sur l'emploi des paramètres de contrôle banalisés.

- les fonctions terminales spécialisées d'accès aux fichiers sont celles proposées par la méthode (modèles préprogrammés).

Bref, les modules utilisés sont des modules COBOL/LCM rédigés en langage objet COBOL.

2. Structure des données

Les données transmises par les différents modules du compilateur sont composées d'un couple (préfixe, donnée).
Le préfixe définit certains attributs de la donnée.
Exemple : TOKEN

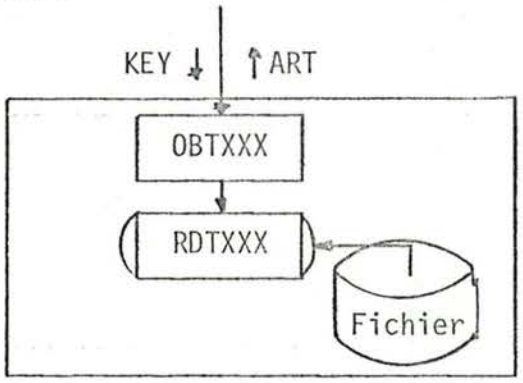
préfixe : n° de ligne dans le programme-source
longueur en caractères
position de début (du premier caractère) dans la ligne
....

donnée : chaîne de caractères.

3. Accès aux tables externes

Le compilateur consulte trois tables externes : pour chaque article de ces tables, il existe une clé d'accès unique, permettant l'accès direct.
Les tables sont stokées dans des fichiers triés par ordre de clé croissante. En début d'exécution, les tables sont chargées en mémoire centrale.
Cette technique nous permet un accès plus rapide aux articles des fichiers, étant donné qu'ils sont relativement petits (en tout $\pm$ 200 enregistrements de 80 caractères). D'autre part, la lecture séquentielle et la recherche en table donne une adaptabilité plus grande vis-à-vis du compilateur COBOL par rapport aux techniques d'accès direct.
Toute fonction de consultation d'une table externe sera composée de deux modules : les modules OBTXXX et RDTXXX.
Au chargement le module OBTXXX appelle le module RDTXXX de lecture séquentielle du fichier (fonction terminale) et met les articles en table.
A chaque demande d'article, le module OBTXXX reçoit une clé d'accès, effectue une recherche en table et transmet l'article demandé.

Fonction de consultation :



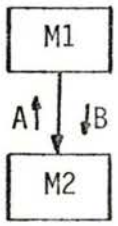
4. Conventions pour la représentation des "diagrammes d'appel" (3)(4)(5)

Un diagramme d'appel est un schéma représentant les relations entre différents modules.
Les modules représentant des fonctions essentielles sont encadrés d'un rectangle(▭), les fonctions terminales (d'entrée/sortie) sont encadrées (▯)
Les fichiers physiques sont représentés ⌐▭ s'ils sont en "input"

⌐▭ s'ils sont en "output"

Les relations "module-appelant - module-appelé" sont matérialisées par une flèche dans le sens module appelant vers module appelé.
Nous y ajoutons le nom des arguments transmis par les modules et leur sens de transmission.

Exemple :



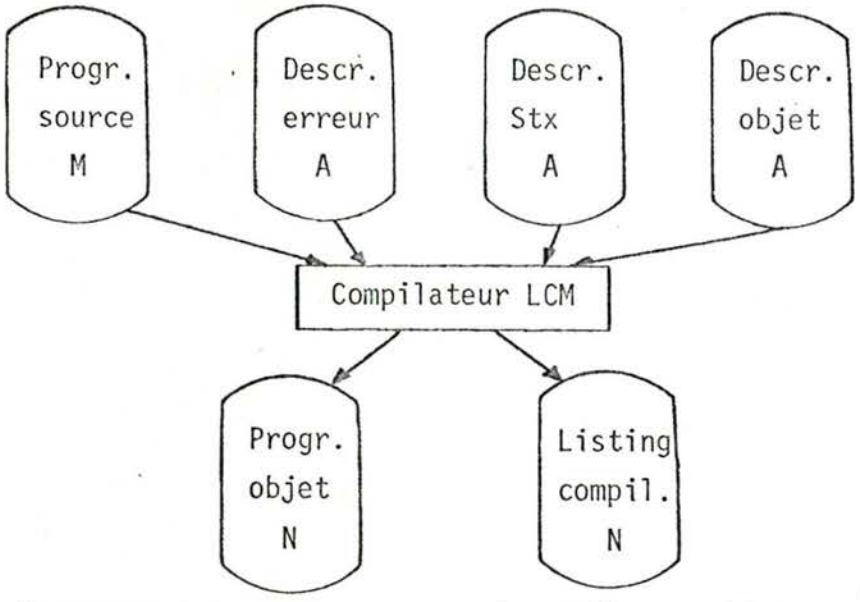
Signification : le module M1 appelle le module M2
le module M1 transmet en guise d'argument à M2 le paramètre B.
le module M2 "renvoie" l'argument A au module M1.

Le fait d'employer la méthode MEMO-PROGES nous amène à transmettre à chaque appel le paramètre global banalisé U-EXT comprenant l'indicateur d'appel U-SW1 et l'indicateur de retour U-SW2.
De manière à ne pas alourdir la représentation des diagrammes d'appel, nous considérons comme implicite la transmission du paramètre U-EXT.
Comme mentionné plus haut, le paramètre des informations globales à tous les modules (exceptés les modules entrée/sortie), le "COMMON-BLOC" est aussi un argument transmis lors des appels de module. Nous le considérons également implicitement transmis pour la représentation des diagrammes d'appel.

B. Structure générale du compilateur

1. Diagramme des flux externes

A partir du fichier "programme-source" et à l'aide des fichiers "Descripteurs syntaxiques", "Descripteurs objet " et "Messages d'erreurs", le compilateur produit systématiquement "un programme-objet", un "listing-compilation" et un "listing-map".



Programme-source : programme à compiler en COBOL/LCM
Descripteurs des messages d'erreur : fichier contenant les textes des messages d'erreur
Descripteurs syntaxiques : fichiers contenant les descripteurs syntaxiques
Descripteurs objet : Fichier contenant les prototypes de traduction des différentes phases LCM.

Programme-objet : Programme compilé comportant :

- les phrases COBOL du programme source inchangées
- les phrases LCM sous forme de lignes de commentaires
- la traduction des phrases LCM en COBOL.

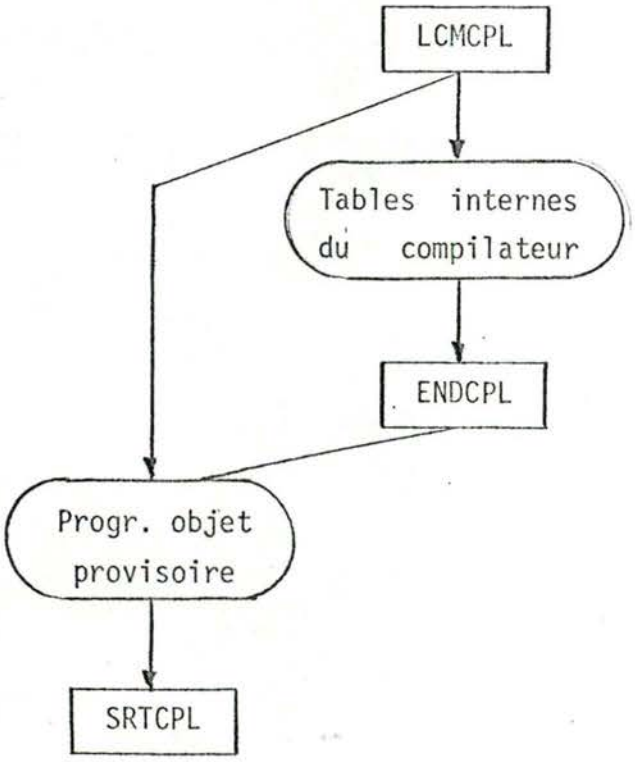
Listing-compilation : Listing du programme-source , chaque ligne étant suivie s'il y a lieu du ou des messages d'erreur la concernant.

2. Découpe en phrases

Les traitements effectués par le compilateur peuvent se subdiviser en 3 phases successives :

1. Le traitement du programme source (LCMCPL)
2. La génération de compléments (ENDCPL)
3. Le tri du programme-objet provisoire (SRTCPL)

Il existe des interfaces entre ces 3 phases :



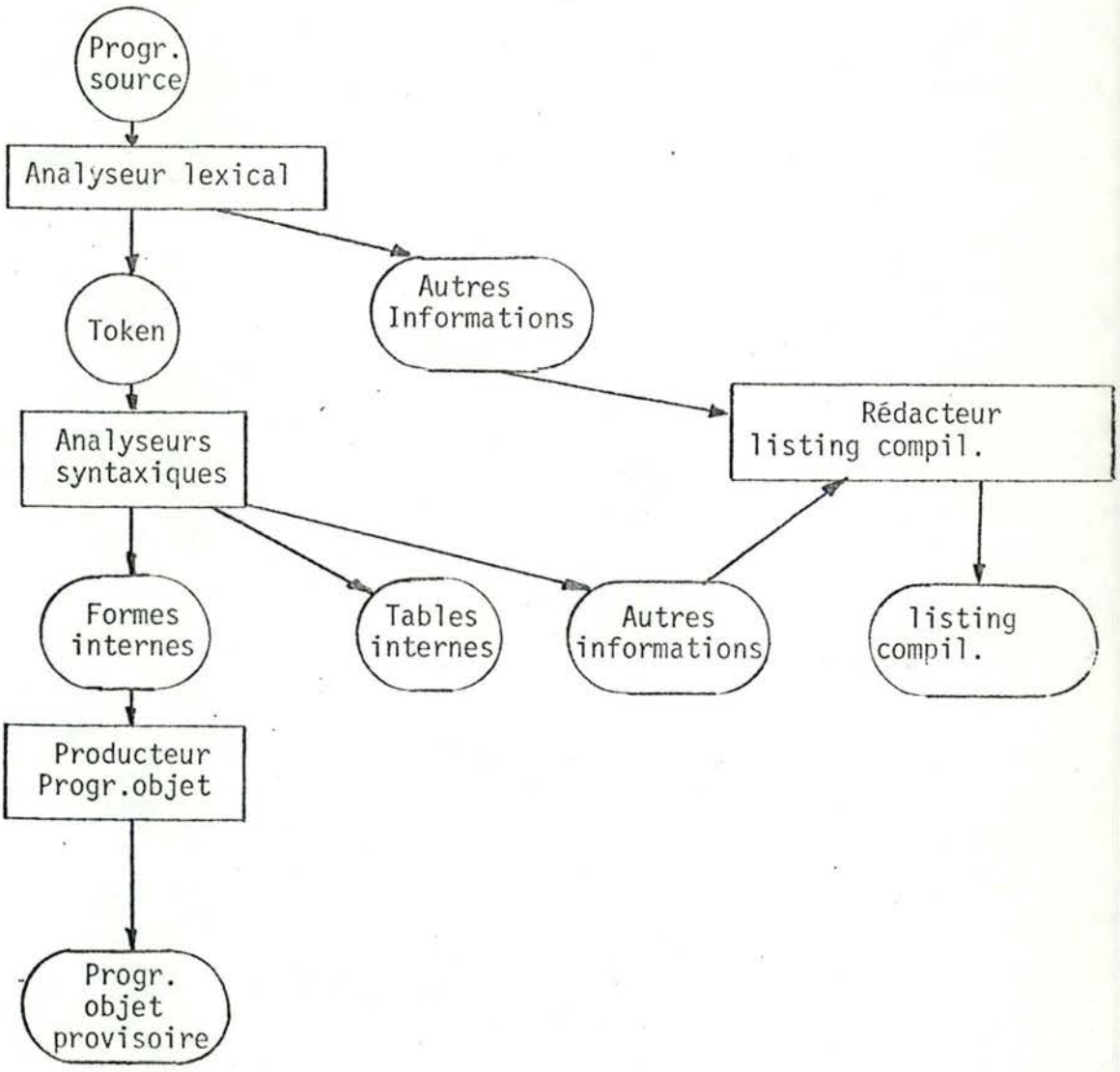
3. Structure logique des différentes phases

a. Phase I : traitement du programme-source

Le traitement du programme source a pour but de produire un texte-objet provisoire et les tables internes nécessaires à la génération des compléments.

Conceptuellement, il se subdivive en quatre fonctions :

1. Analyseur lexical
2. Analyseurs syntaxiques
3. Producteur de programme-objet
4. Rédacteur du listing compilation

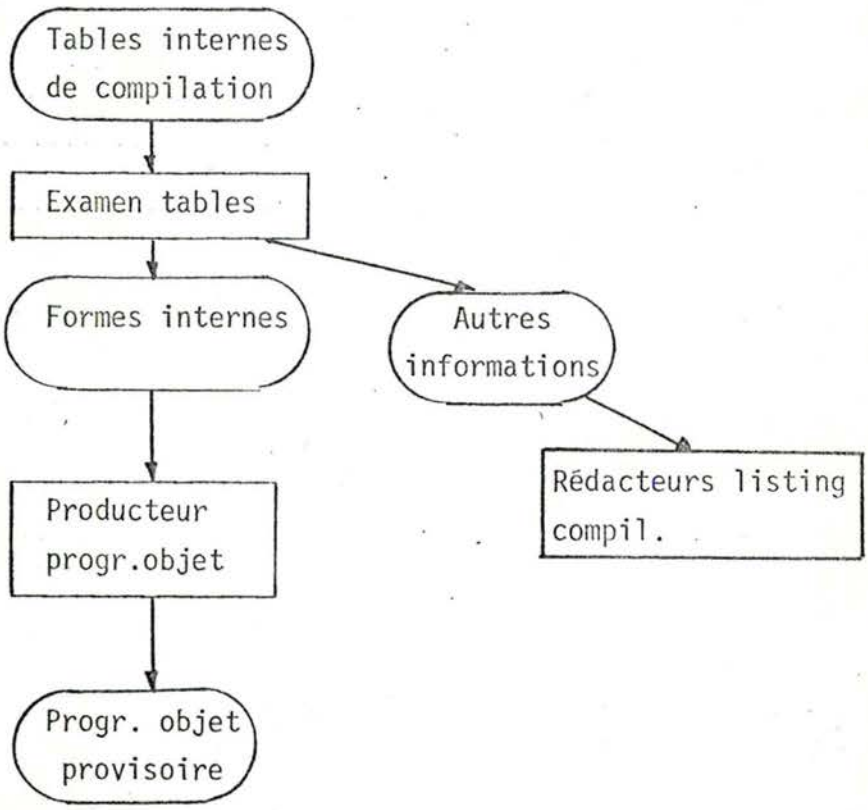


b. Phase II : Générateur de compléments

Le générateur de compléments examine les tables internes constituées à la phase I du compilateur et complète le programme objet provisoire s'il y a lieu.

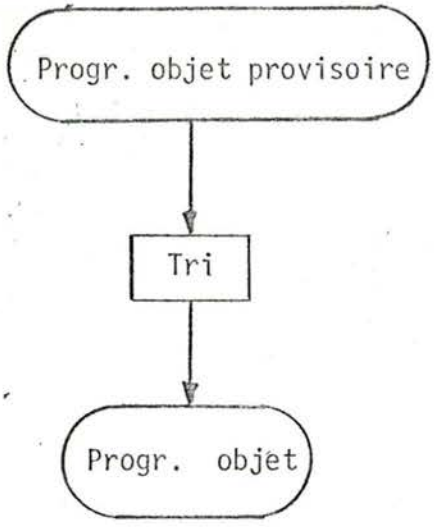
Conceptuellement, il se subdivise en :

- Examen des tables internes et production de formes internes
- Producteur de programme-objet
- Rédacteur du listing compilation



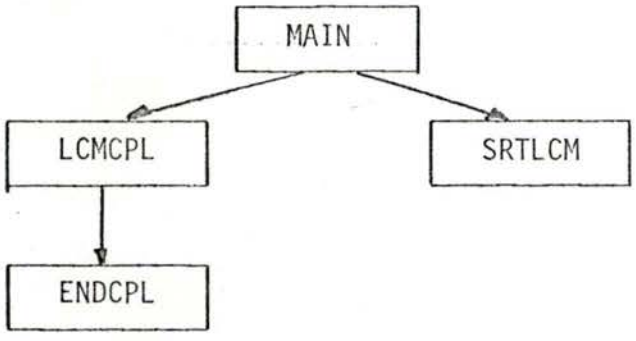
c. Phase III : Tri du programme-objet provisoire

Cette phase est uniquement un tri d'un fichier - le programme objet provisoire donnant un programme objet.



4. Description de l'unité de traitement

L'unité de traitement est composée d'un module directeur (MAIN) activant successivement les phases I et III du compilateur (phases de traitement du programme-source et de tri du programme objet provisoire).
La phase II est quant à elle activée par la phase I lorsque celle-ci est terminée.
Le diagramme des appels est le suivant :



Commentaires relatifs aux différents modules :

MAIN : module directeur de la méthode MEMO-PROGES -
effectue la lecture du nom du fichier-source et transmet ce nom par l'intermédiaire de U-EXT (interface banalisé)

LCMCPL : cfr supra

SRTLCM : cfr supra

ENDCPL : cfr supra.

5. Liste des informations contenues dans le COMMON-BLOC

- CTR-ERROR : Compteur du nombre d'erreurs "fatales"
- TABLE-SGMT: Table des codifications des différents segments
  - Codification du segment
  - nom du segment
  - statut du segment :
    le segment peut être obligatoire ou facultatif
    dans le programme source
    dans le programme objet.
  - nombre de lignes objet générées dans ce segment (hauteur de la pile)
  - nombre de fois où le segment a été rencontré dans le programme.
- ACTUAL-LANGAGE : langage traité "actuellement" par le compilateur
- ACTUAL-SGMT : Numéro de segment COBOL et LCM "actuels"
- TABLE DE PARAMETRES :
  - . nom du segment COBOL "actuel"
  - . nom du segment LCM"actuel"
  - . nom du segment COBOL précédent
  - . nom du segment LCM précédent
  - . nom du programme (du module compilé) (àIENTRY)
  - . nom du premier argument de la phrase àIENTRY
  - . nom du deuxième argument de la phrase àIENTRY
  - .

C. Analyseur lexical

L'analyseur lexical est une "sous-routine" des analyseurs de séquences terminales LCM et COBOL. Chaque fois qu'il est appelé, il fournit le "token" suivant dans le programme source. D'autre part, il effectue des vérifications au niveau de la structure lexicale de la ligne COBOL/LCM et envoie chaque ligne lue à destination des rédacteurs de listing compilation et de programme-objet provisoire.

1. Composition des unités d'information :

LINE-READ :-(chaîne de 120 caractères) ligne lue dans le programme source en format COBOL DEC 20 (qui est différente du format norme COBOL)

STD-LINE :-ligne du programme source en format norme COBOL

LINE-REC V : - préfixe : . numéro ligne (numérotation continue)
. numéro de la position du premier caractère dans la ligne
. numéro de la position du dernier caractère de la ligne.
- contenu : - ligne du programme source

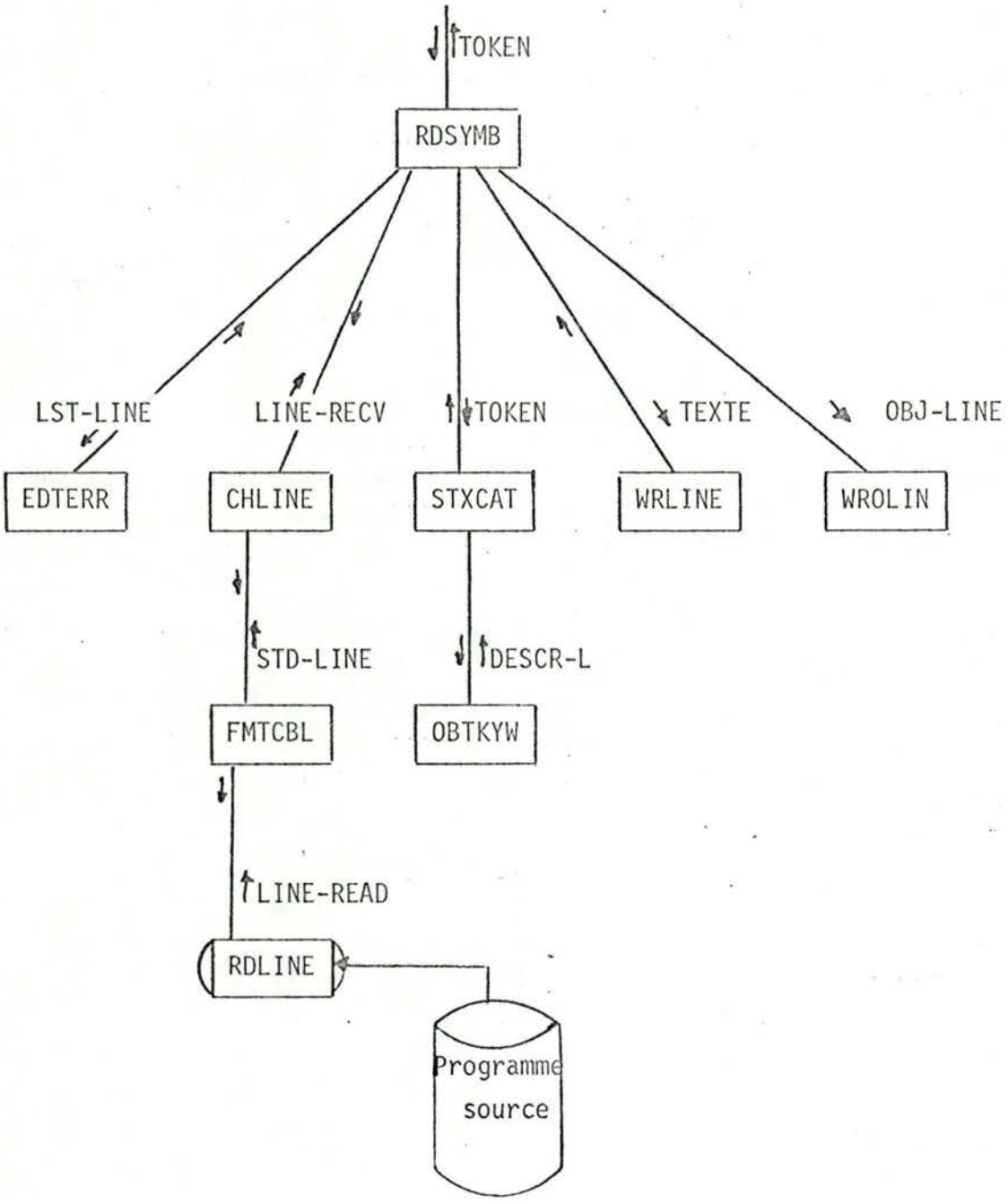
TOKEN : - préfixe : - numéro ligne du premier caractère du symbole
- numéro ligne du dernier caractère du symbole
- position dans la ligne du premier caractère
- longueur du symbole en caractères.
- catégorie syntaxique : . mot clé COBOL (mot réservé)
. mot COBOL
. littéral COBOL non numérique
. nombre entier sans signe
. caractère spécial.
- séparateurs:
SEPAR (1) indique si le symbole
n'est pas suivi d'un blanc
est suivi d'au moins un blanc
est en fin de ligne.
SEPAR (2) indique si le symbole est suivi d'un ";" ou d'une ","
SEPAR (3) indique si le SEPAR (2)
n'est pas suivi d'un blanc
est suivi d'au moins un blanc
est en fin de ligne.
- contenu : symbole

LST-LINE : - numéro du message d'erreur
- zone où si l'erreur le permet, le module de lecture met le symbole erroné.

TEXTE et OBJLIN : - zone de préfixe mise à blanc
(cfr rédacteur listing compilation)
(cfr rédacteur programme objet)
- STD-LINE

DESCR-L : - contenu : symbole.

2. Diagramme des appels



3. Commentaires relatifs aux différents modules

RDLINE : Fonction terminale de lecture du fichier "programme-source"
FMTCBL : "Formattage" de la ligne "format DEC" en format COBOL(norme)
CHLINE : numérotation des lignes et détermination du premier et dernier caractères non blancs dans la ligne.
RDSYMB : Extraction des "token" dans la ligne COBOL/LCM
Vérification de la structure lexicale de la ligne COBOL/LCM : (coupure de littéraux, lignes blanches, signes de continuation, découpage de la ligne d'écriture).
Transmission de la ligne aux modules de rédaction du listing compilation et du programme objet.
STXCAT : Détermination de la catégorie syntaxique du "TOKEN"
OBTKYW : Recherche dans la table des mots réservés COBOL : le symbole est-il un mot réservé COBOL ?
WRLINE : cfr. Rédacteurs de listing
WROLIN : cfr. Rédacteur du programme-objet.

D. Analyseur syntaxique

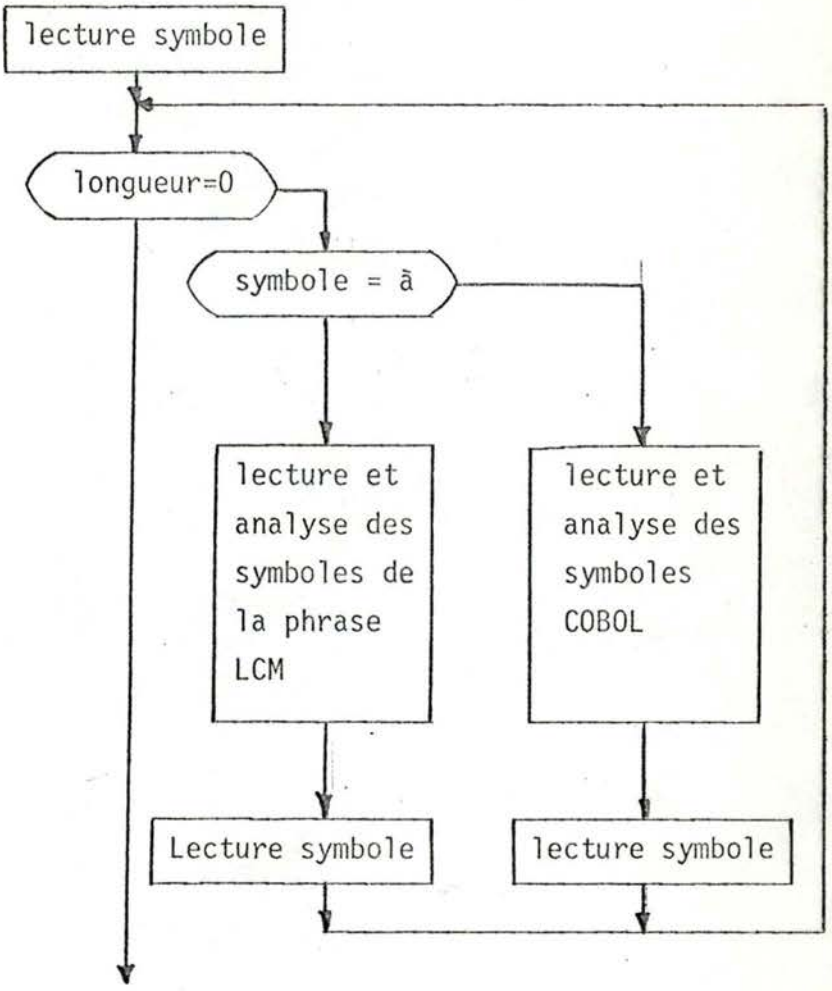
Le programme source est composé alternativement de phrases COBOL et de phrases LCM. Les phrases sont des suites de "tokens" et sont disjointes.
Le problème de l'analyse se décompose comme ceci :

1. L'analyse du type de phrase.
2. L'analyse d'une phrase COBOL
3. L'analyse d'une phrase LCM.

1. Analyseur du type de phrase

Toute phrase LCM commence par le symbole " ", ce qui nous permet de déterminer la transition COBOL-LCM et"d'activer" l'analyseur LCM.
L'analyseur LCM traite une phrase LCM et détermine la transition LCM-COBOL et de "désactive".

Le système de lecture des symboles (Appels à l'analyseur lexical) est tel que l'on possède toujours un symbole d'avance, c'est-à-dire que les analyseurs LCM et COBOL lisent un symbole en plus de ceux qu'ils ont reconnus. L'analyseur lexical renvoie un symbole de longueur nulle lorsqu'il n'a plus de symbole à lire.



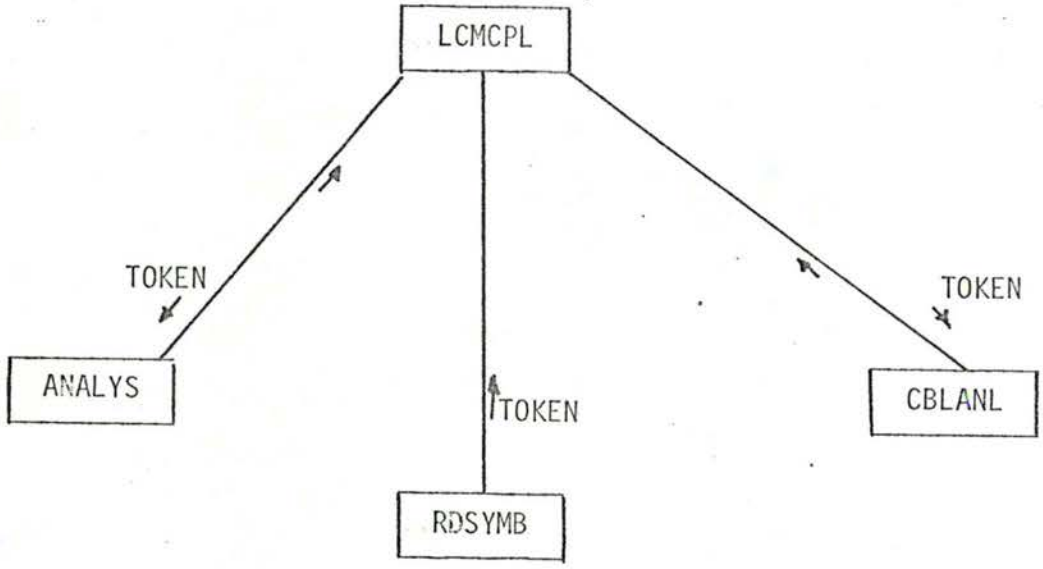
L'analyse du type de phrase est articulée autour des modules

LCMCPL : analyseur du type de phrase

ANALYS : analyseur d'une phrase LCM

CBLANL : analyseur d'une phrase COBOL

Le diagramme des appels est le suivant :



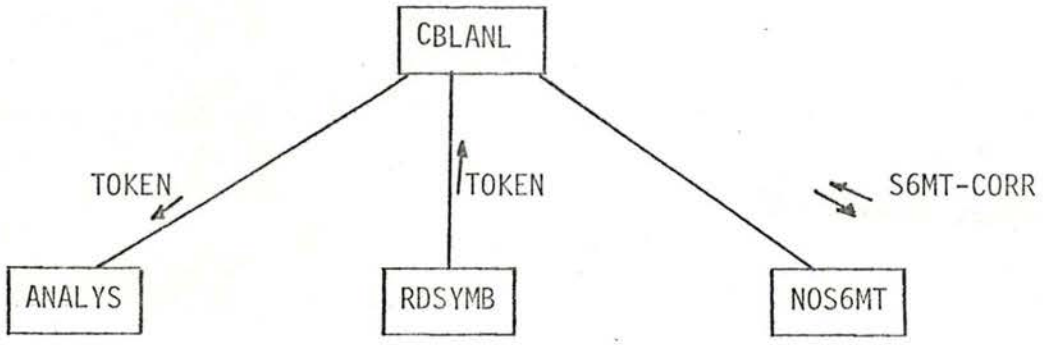
2. Analyseur d'une phrase COBOL

Les phrases COBOL sont "ignorées" du compilateur (c'est-à-dire recopiées telles quelles dans le programme-objet) exceptés les titres de sections et de divisions.
Ces titres de sections ou divisions seront dès lors considérés comme appartenant au langage LCM (donc passés à ANALYS).

1. Composition des unités d'information

TOKEN : cfr supra
S6MT-CORR : - nom de segment COBOL
- numéro du segment COBOL

2. Diagramme des appels



3. Commentaires relatifs aux différents modules

CBLANL : module d'analyse des phrases COBOL
Sélectionne les phrases de titre de section ou de division
ANALYS : cfr supra
RDSYMB : cfr infra
NOSGMT : module de mise à jour des informations segment actuel - segment précédent dans le COMMON-BLOC.

3. Analyseur de phrases LCM

L'analyseur de phrases LCM effectue des traitements au niveau
- des symboles de la phrase
- de la phrase tout entière.

a. Traitements au niveau des symboles

Pour chaque symbole de la phrase, l'analyseur vérifie la concordance descripteur syntaxique - symbole rencontré et crée une forme interne s'il y a lieu.

b. Traitements au niveau de la phrase

L'analyseur effectue la vérification du contexte de la phrase (contexte = fait d'appartenir à une section ou une division).
Si la phrase est un titre, il s'agit de mettre à jour la table des paramètres contenue dans le COMMON-BLOC.

Si la phrase est un "OPEN" , "ICALL", "CLOSE" ou "PREPARE", il s'agit de mémoriser dans la table interne des modules appelés le fait d'avoir rencontré cette phrase.

1. Composition des unités d'information

TOKEN : cfr supra

PARSER-DES : descripteur d'analyse syntaxique

clé d'accès : - OPER-NAME : nom du premier mot de la phrase analysée, ou nom du graphe ou du sous-graphe.

- numérotation :.continue à partir de 00 pour un graphe syntaxique
.continue à partir de 01 pour un sous-graphe syntaxique.

contenu : a) pour les descripteurs ØØ

- description du contexte de la phrase : liste des segments où la phrase peut se trouver.
- description de la mise en page de la phrase : place de la phrase dans la ligne COBOL.
- description des traitements à effectuer après l'analyse de la phrase :
  1. s'il s'agit d'un titre : mise à jour des paramètres actuels-précédents dans le COMMON-BLOC.
  2. possibilité de faire à partir des formes internes des variables globales dans le COMMON-BLOC
  3. s'il s'agit d'un OPEN-CALL-CLOSE. Mémorisation de la phrase.

b) pour les descripteurs "nn" :

1. Descripteurs associés à la description d'un élément syntaxique :

- description de l'élément syntaxique (token)
  - . Catégorie syntaxique : mot clé COBOL, mot COBOL, entier, littéral, caractères spéciaux, signe de ponctuation ".".
  - . Valeur du token, si celui-ci doit en avoir une bien déterminée.
  - . Séparateurs suivant le token , soit :

    SEPAR (1) et (3) - pas de blanc
    - un blanc ou plus
    - fin de ligne.

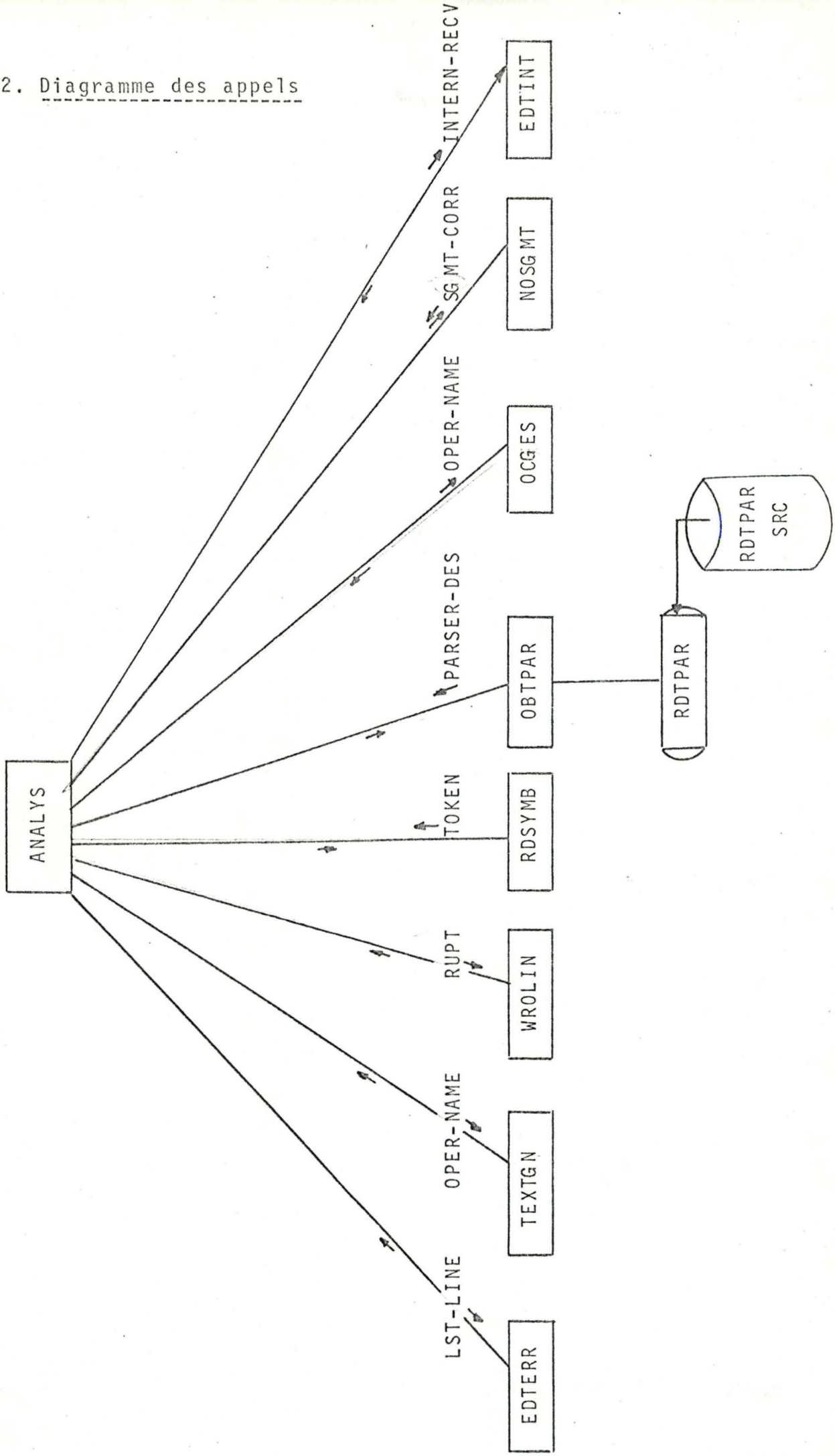
    SEPAR (2) - soit inexistant
    - soit ";" ou ",".

- Traitements "OK" et "KO"
  - . numéro du message d'erreur ou d'avertissement à produire.
  - . numéro de la forme interne à produire
  - . contenu de la forme interne
    - soit le symbole analysé
    - soit un texte.

2. Descripteurs associés au branchement vers un sous-graphe.

- nom du sous-graphe dans lequel se poursuivra le cheminement
- numéro du descripteur suivant dans le graphe.

2. Diagramme des appels



3. Commentaires relatifs aux différents modules

ANALYS : module d' analyse syntaxique
- effectue le parcours du graphe syntaxique en créant des formes internes s'il y a lieu.
- donne les informations de mise en page au module d'écriture du programme-objet.

EDTERR : module d'édition des erreurs. cfr supra.
TEXTGN : module de production du programme-objet provisoire. cfr supra.
WROLIN : module de mise en page du programme-objet provisoire. cfr supra.
RDSYMB : cfr infra.
OBTPAR : module d'accès aux tables internes contenant les descripteurs syntaxiques.
RDTPAR : fonction terminale d'accès au fichier contenant les descripteurs syntaxiques (RDTPAR.SRC)
OCGES : module de création et de consultation (cfr supra) d'une table interne du compilateur renfermant les informations nécessaires à la gestion des "OPEN", "CLOSE" et "PREPARE".
NOS6MT : cfr infra.
EDTINT : module d'insertion et de consultation de la table des formes internes.

E. Producteur du listing-compilation

Le listing-compilation est composé du programme-source, chaque ligne du programme-source étant , s'il y a lieu, suivie du (des) message(s) d'erreurs la concernant.
La structure de l'appel du module "RDSYMB" (module extracteur des tokens du programme-source) est telle que celui-ci envoie d'abord la ligne du programme-source suivie des lignes de messages d'erreurs la concernant.

a. Composition des unités d'information

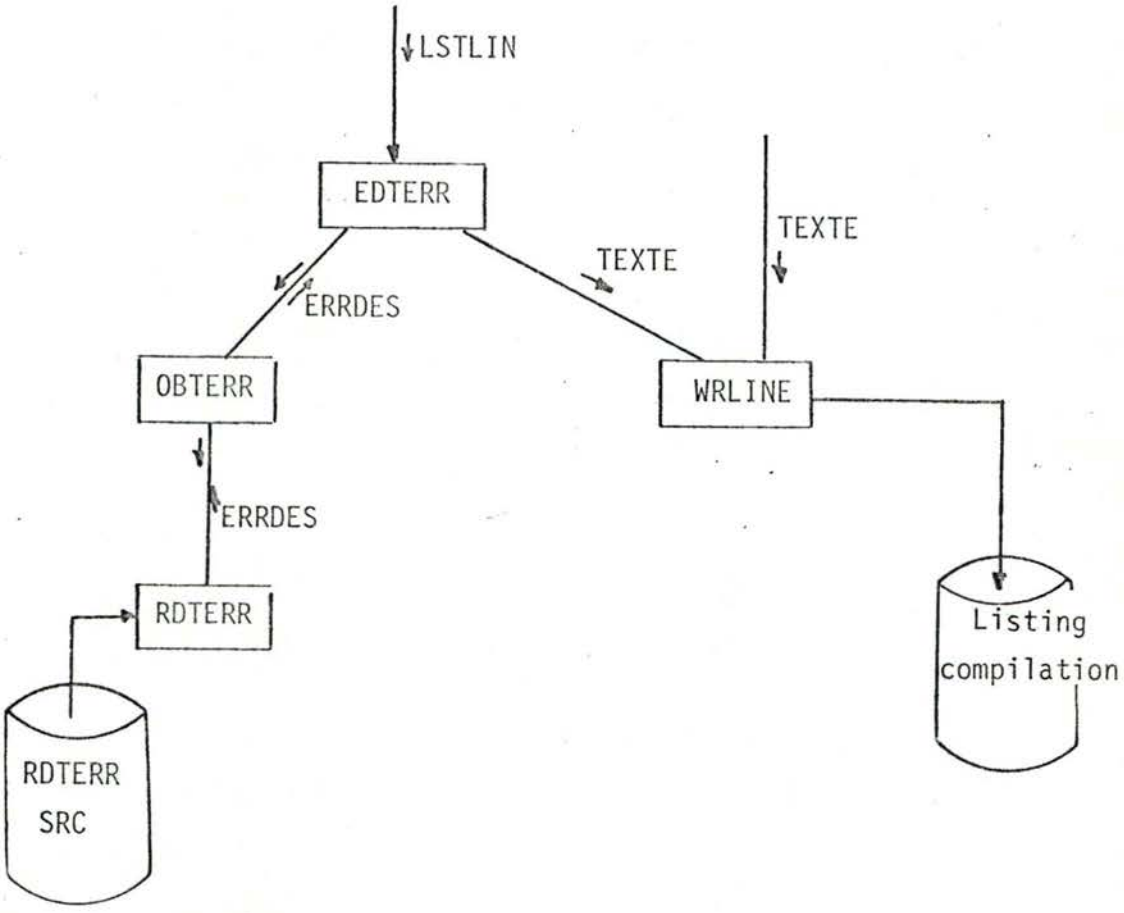
LSTLIN : - numéro du message d'erreur.
Le numéro du message permet de distinguer s'il s'agit d'une erreur fatale ou d'un avertissement.

- zone, où lors de la détection de l'erreur, on a pu mettre un symbole erroné facilitant la correction par le programmeur.

ERRDES : - numéro du message d'erreur
- texte du message d'erreur

TEXTE : - préfixe = blanc si ligne du programme source
= numéro de message si ligne d'erreur.
- texte du programme-source ou du message d'erreur.

b. Diagramme des appels



c. Commentaires relatifs aux différents modules

EDTERR : module d'édition des messages d'erreurs.
Effectue la concaténation - du type de message d'erreur
- du texte du message d'erreur
- du symbole concerné par l'erreur.

OBTERR : module d'obtention du texte d'un message d'erreur sur base d'un numéro de message

RDTERR : fonction terminale de lecture du fichier des messages d'erreurs

WRLINE : fonction terminale d'écriture sur le fichier listing compilation.

F. Générateur du programme-objet

1. Producteur du texte-objet provisoire

Le producteur du texte-objet provisoire est une "sous-routine" des analyseurs syntaxiques et du générateur de compléments.

Celui-ci se compose de deux parties :

- le générateur de lignes-objets
- le rédacteur du programme-objet.

a) Le générateur de lignes-objets

Le générateur de lignes-objets produit une série de lignes de texte-objet à destination du rédacteur.

a. Composition des unités d'informations

OPER-NAME : nom de l'opération : clé d'accès à un ensemble de descripteurs de la (des) ligne(s) objet (prototypes).

OBYDES : Descripteur d'une ligne objet à générer.

clé d'accès : - nom de l'opération
- numéro de ligne (numérotation continue pour une opération donnée).

préfixe : - codification permettant la génération d'une ligne conditionnellement à l'existence ou à la non-existence de certains paramètres
  . dans une forme interne
  . dans la table des paramètres du COMMON-BLOC.
- codification permettant la génération de la ligne - dans un segment COBOL et LCM
  . spécifié

. dans le segment "courant"
. dans le segment "précédent"

donnée : texte-prototype

Le texte prototype est un schéma (image) de la traduction d'une instruction.
Celui-ci contient une chaîne de caractères qui à l'exception des méta-symboles , sera recopiée dans le programme objet.
Exemple de texte prototype - sans métasymboles.
Traduction de l'instruction àCORPUS SECTION

Texte prototype : U-CORPUS SECTION
G-CORPUS

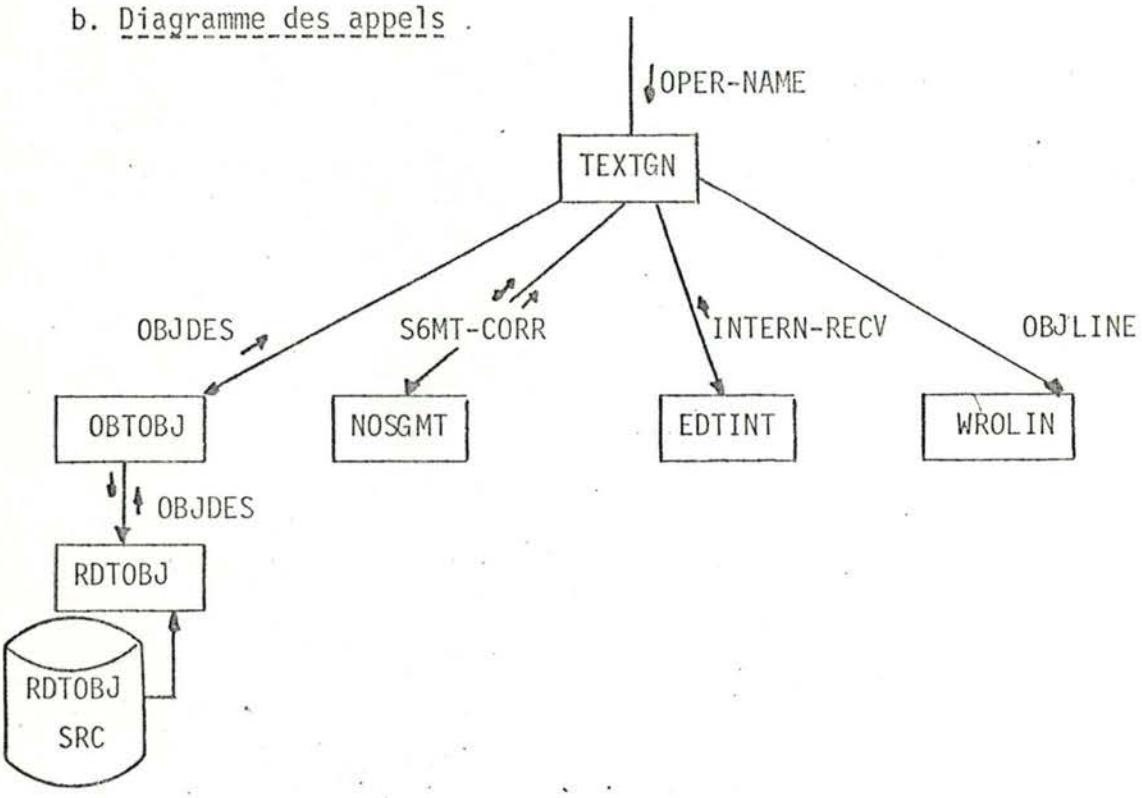
Les métasymboles permettent de faire référence à des symboles créés par le programmeur. Ceux-ci peuvent se trouver dans la table des paramètres du COMMON-BLOC ou dans une forme interne.
Exemple de texte prototype - avec métasymbole.
Traduction de l'instruction :
à ICALL <MODULE> (ARG1,ARG2)
L'analyseur syntaxique garnit les 3 formes internes suivantes :

I1 : < MODULE >
I2 : < ARG1 >
I3 : < ARG2 >

Texte prototype : MOVE U-SWM TO U-SW1 U-SW2
CALL "##I1" USING U-EXT #I2#I3

INTERN-RECV : cfr supra
SGMT-CORR : - numéro de segment LCM ou COBOL
- nom du segment
OBJLIN : cfr supra

b. Diagramme des appels



c. Commentaires relatifs aux différents modules

TEXTGN : module de génération de lignes de texte-objet pour une phrase donnée.

OBTOBJ : module de consultation de la table des prototypes de traduction (OBYDES)

RDTOBJ : fonction terminale de lecture du fichier contenant les prototypes de traduction (RDTOBJ.SRC)

NOSGMT : module donnant la correspondance entre le numéro du segment COBOL/LCM actuel et le nom du segment COBOL/LCM.

EDTINT : module de création (cfr analyseur) et d'obtention d'une forme interne.

WROLIN : module de mise en page du programme-objet provisoire (cfr supra).

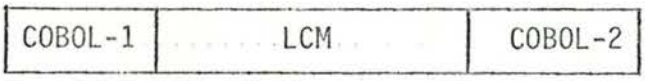
b) Rédacteur du programme-objet

La fonction du rédacteur du programme-objet est de "mettre en page" les différentes lignes du programme-objet.

Nous avons considéré le programme-objet comme une "copie" du programme-source avec néanmoins (!) certaines modifications.

Les phrases LCM sont copiées sur des lignes de commentaires et suivies de leur traduction en COBOL.

Ceci demande la mise en oeuvre de mécanismes particuliers.
Ainsi, pour une ligne du programme source composée alternativement d'une séquence COBOL-LCM-COBOL :



le texte se présentera comme ceci :

1. Description des unités d'information

OBJLIN : préfixe : - numéro de segment COBOL
- numéro de segment LCM
- numéro de ligne dans le segment COBOL (numérotation continue)
- numéro de la ligne dans le segment LCM(numérotation continue)
- numéro de la ligne dans le programme-source (référence)
- numéro de la sous-ligne par rapport à une ligne du programme-source

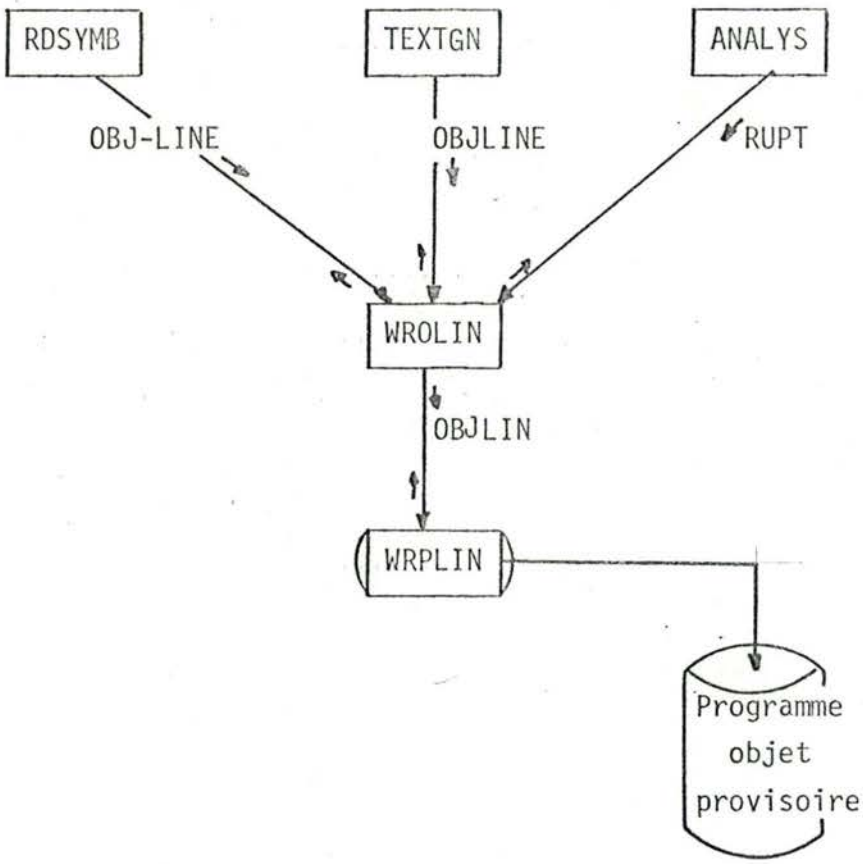
Remarque : Toutes ces informations fournissent la clé de tri du programme objet provisoire.

- langage - COBOL
- LCM
- COBOL généré
- numérotation continue dans le programme objet.

donnée : - ligne générée.

RUPT : Information permettant de faire la coupure des lignes (ruptures de langage)
- type rupture soit LCM/COBOL ou COBOL/LCM
- position de la rupture dans la ligne source
- numéro de la ligne source concernée.

2. Diagramme d'appel



3. Commentaires relatifs aux différents modules

ANALYS : analyseur syntaxique détectant les ruptures de langage

TEXT6N : générateur de texte produisant des lignes objets (traduction d'une phrase LCM)

RDSYMB : analyseur lexical livrant les symboles du programme source.

2. Générateur de compléments

Le générateur de compléments a pour but de produire des lignes COBOL considérées comme implicites dans le programme-source mais obligatoires dans le programme objet.

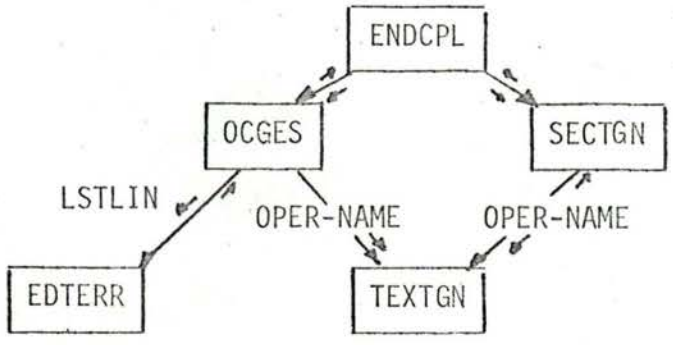
Ces lignes sont relatives :
- aux SECTIONS et DIVISIONS COBOL et LCM qui sont nécessaires à la gestion implicite de l'itération.
- aux initialisation et terminaisons de l'indicateur d'appel (OPEN et CLOSE implicites)

Il effectue d'autre part des vérifications relatives aux sections et divisions (un seul segment de chaque type et segments COBOL dans l'ordre défini par la norme) ainsi qu'aux "OPEN", "CLOSE" et "PREPARE".

a. Composition des unités d'informations

OPER-NAME : nom de l'opération à effectuer
Exemple : IMPLCALL : génération d'un OPEN et CLOSE implicite . cfr supra.

b. Diagramme d'appels



c. Commentaires relatifs aux différents modules

ENDCPL : module principal de la génération des compléments
SECTGN : module déterminant les sections et divisions manquantes
OCGES : module déterminant les "OPEN" et "CLOSE" implicites
TEXTGN : cfr supra.

Conclusions

Ce travail avait pour but de réaliser un des outils logiciels associés à MEMO-PROGES , à savoir l'élaboration d'une version d'un compilateur ayant un programme source rédigé en COBOL-LCM et produisant un programme objet en COBOL.

Le langage LCM possède quelques particularités :

1. Il est inclus dans un langage "hôte" , ici le COBOL et possède une syntaxe apparentée au COBOL. Il pourrait être inclus dans un autre langage "hôte".
2. Il est susceptible de recevoir des extensions , des modifications.
3. Sa traduction est effectuée en COBOL, mais pourrait être faite dans un autre langage.

Dans l'élaboration du compilateur, nous avons cherché une certaine indépendance (c'est-à-dire que nous avons cherché à minimiser les efforts de reprogrammation en cas de modifications). Cette indépendance est obtenue au moyen de tables descriptives.

- indépendance vis-à-vis du langage "hôte" et de la structure des programmes rédigés dans ce langage : l'analyse de la structure basée sur une table décrivant la segmentation est une solution générale : en effet, le langage LCM est un langage segmenté, tout comme COBOL. Toute insertion dans un autre langage "hôte" amènera aussi une segmentation, tout au moins pour le LCM.
- indépendance vis-à-vis de la syntaxe du LCM : celle-ci est obtenue par l'emploi d'une table externe : les descripteurs syntaxiques.
- indépendance vis-à-vis du langage-objet : en effet, la traduction des phrases LCM est décrite également dans une table externe : les prototypes de traduction.

Au point de vue de la réalisation du compilateur, nous avons utilisé les deux principes essentiels du système de programmation de MEMO-PROGES (1), à savoir : - l'interface banalisé
- les fonctions terminales spécialisées.

Grâce à ceux-ci, nous disposons d'une certaine souplesse dans la reconfiguration des programmes :

- le remplacement ou l'ajout de modules est facilité par l'emploi de l'interface banalisé, car nous profitons de la quasi compatibilité des modules (possibilité de coupler entre eux des modules quelconques, sous réserve de quelques restrictions valables pour tout module indistinctement). Ainsi, des modules d'analyse sémantique et de structuration du programme pourraient être ajoutés.
- l'emploi des fonctions terminales spécialisées donne une indépendance totale vis-à-vis du support des tables externes : celles-ci peuvent être par exemple des tables programmées, des fichiers séquentiels traités comme tables en mémoire centrale, des fichiers séquentiels indexés. Pour modifier le support, il suffit d'incorporer au programme la fonction terminale adéquate.

Nous avons aussi utilisé deux techniques courantes dans l'élaboration d'un compilateur, à savoir [6] [8]

- la déconnection entre les analyseurs lexical et syntaxique. Celle-ci est obtenue grâce à un interface : le TOKEN. (symbole extrait du programme source)
- la déconnection entre l'analyseur syntaxique du texte source et le générateur du programme-objet : celle-ci est obtenue par un autre interface : la table des formes internes (contenant uniquement les informations de la phrase LCM nécessaires à la génération de sa traduction).

La technique de génération d'un texte objet provisoire et de tri de celui-ci paraît, à postériori, superflue : moyennant quelques modifications, il serait possible de produire directement un programme objet définitif : en effet, dans notre version, d'une part l'analyse et la vérification de la structure du programme source, d'autre part la création d'une structure dans le programme-objet sont disséminées dans l'analyseur syntaxique, le générateur de texte source, le générateur de compléments et le tri du programme objet provisoire.
Une prochaine version devrait, à l'aide de modules spécialisés , résoudre plus élégemment ce problème.

BIBLIOGRAPHIE

(1) A. CLARINVAL — MEMO-PROGES : Méthode modulaire de programmation pour l'informatique de gestion.
Thèse de doctorat, Namur, le 24 octobre 1977

(2) A. CLARINVAL — MEMO-PROGES : Définition des langages "LDF" et "LCM"
(à paraître)

(3) W.P. STEVENS, G.J. MYERS, L.L. CONSTANTINE
Structured design
I.B.M. Syst. J. 13/2 , 1974

(4) D.L. PARNAS — A technique for software module specification with exemples.
Comm. ACM. 15/5 May 1972

(5) J.F. STAY — HIPO and integrated program design
I.B.M. Syst. J. 15/2 , 1976

(6) D. GRIES — Compiler construction for digital computers
John Wiley, 1971

(7) FOSTER J.M. — Automatic Syntactic Analysis
Elsevier 1970

(8) HORNING J.J. — A compiler generator
Prentice Hall, 1970

(9) INGERMAN P.Z. — A syntax oriented translator
Acad. Press., 1966

(10) M.BLOCK et C. BRISSEAU
Analyse syntaxique par cheminement sur un graphe
Méthode et mise en oeuvre.
RIRO, 4e année, B-1, 1970, p.3-20

(11) L. BOLLIET L'écriture des compilateurs - I
RFTI - chiffres vol.9, n°1, 1966, p47-73